GCA S3 03

Operating System

SEMESTER III

BACHELOR OF COMPUTER APPLICATION

BLOCK 1



KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Experts

Prof. Anjana Kakati Mahanta,	Gauhati University
Prof. (Retd.) Pranhari Talukdar,	Gauhati University
Dr. Jyotiprokash Goswami	Assam Engineering College

Course Coordinator

Dr. Sanjib Kr. Kalita,	KKHSOU
Dr. Tapashi Kashyap Das,	KKHSOU
Sruti Sruba Bharali	KKHSOU

SLM Preparation Team

Units	Contributor
1	Dr. Ridip Dev Choudhury, IDOL, Gauhati University
2	Dr. Ridip Dev Choudhury, IDOL, Gauhati University
3	Dr. Meenakshi Gogoi, GIMT
4	Sruti Sruba Bharali, KKHSOU
5	Dr. Ridip Dev Choudhury, IDOL, Gauhati University
6	Irani Hazarika, Gauhati University
7	Dr. Sanjib Kumar Kalita, KKHSOU
8	Dr. Khurshid Alam Borbora, IDOL, Gauhati University

Editorial Team

Content	Dr. Sanjib Kumar Kalita, KKHSOU
Language	(Retd.) Prof & HOD English, Robin Goswami, Cotton College

Structure, Format & Graphics: Sruti Sruba Bharali, KKHSOU

June, 2018

This Self Learning Material (SLM) of the Krishna Kanta Handiqui State Open University is made available under a Creative Commons Attribution-Non Commercial-Share Alike 4.0 License (international): http://creativecommons.org/licenses/by-nc-sa/4.0/

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University.

Headquarters : Patgaon, Rani Gate, Guwahati - 781017

Housefed Complex, Dispur, Guwahati-781006; Web: www.kkhsou.in

The University acknowledges with thanks the financial support provided by the **Distance Education Bureau**, **UGC** for the preparation of this study material.

COURSE INTRODUCTION

Operating systems are the basis of all computer systems. Operating systems provide the user a interface to connect with computer hardware. It is the software that manages the computer hardware and software resources. Process management, memory management, I/O management and file system management are some of the important functions of the operating systems.

This course on **Operating System** comprises of fourteen units and seeks to present an overview of different opearting system fundamentals in computer system. This course includes an introduction to operating systems, processes and threads and inter process communication. The concepts of semaphores, scheduling and deadlocks are also introduced in this course. Apart from these, memory management techniques like segmentation and paging are covered here along with I/O management, file systems and operating system security topics. The course also contains an introduction to multiprocessor systems.

The course is divided into two blocks:

Block 1 introduces the different types of operating systems. Concepts of process and threads are included in this block. It also contains concepts like interprocess communication, process scheduling, semaphores and deadlocks.

Block 2 concentrates on memory management techniques of opearting systems. Segmentation and paging concepts are introduced in this block in addition to file systems, I/O management, and security. An introduction to multiprocessor operating system is also provided in this block.

Each unit of these blocks includes some along-side boxes to help you know some of the difficult, unseen terms. Some “EXERCISES” have been included to help you apply your own thoughts. You may find some boxes marked with: “LET US KNOW”. These boxes will provide you with some additional interesting and relevant information. Again, you will get “CHECK YOUR PROGRESS” questions. These have been designed for you to self-check your progress of study. It will be helpful for you if you solve the problems put in these boxes immediately after you go through the sections of the units and then match your answers with “ANSWERS TO CHECK YOUR PROGRESS” given at the end of each unit.

BLOCK INTRODUCTION

This is the *First Block* of the course “Operating System”. This block comprises the following eight units:

Unit - 1 introduces you to the different types of operating systems.

Unit - 2 describes the basics of processes. Topics like process life cycle and process control block are discussed in this unit.

Unit - 3 gives us an introduction to threads. Different types of threads are discussed in this unit.

Unit - 4 introduces us to the basics of interprocess communication. Message passing and shared memory communication are described in this unit.

Unit - 5 deals with process synchronization and process co-ordination. Problem and solutions to race condition, critical section and mutual exclusion are discussed in this unit.

Unit - 6 gives us an introduction to semaphores. Binary semaphores are discussed in this unit.

Unit - 7 introduces us to the basics of scheduling. First In First Out, Round Robin are some of the scheduling algorithms described in this unit.

Unit - 8 deals with deadlocks. Conditions for deadlock and methods for handling deadlock like avoidance and recovery are discussed in this unit.

The structure of Block 1 is as follows:

UNIT 1 : Introduction to Operating System

UNIT 2 : Processes

UNIT 3 : Threads

UNIT 4 : Interprocess Communication

UNIT 5 : Process Synchronization and Co-ordination

UNIT 6 : Semaphores

UNIT 7 : Scheduling

UNIT 8 : Deadlocks

BACHELOR OF COMPUTER APPLICATION

OPERATING SYSTEM

DETAILED SYLLABUS

Block-1

Pages

Unit-1 : Introduction to Operating System **7 – 17**

Operating system, Evolution of Operating Systems- batch system, multi-programmed system, time-sharing system, personal computer system, parallel systems

Unit-2 : Processes **18 – 29**

Concept of process, Process life cycle, Process Control Block (PCB), Creation and termination of Process

Unit-3 : Threads **30 – 39**

Concept of Threads, Uses of Thread, Design Issues of Thread, Thread Management

Unit-4 : Interprocess Communication **40 – 47**

Interprocess Communication, Message passing model, shared memory model

Unit-5 : Process synchronization and Co-ordination **48 – 56**

Concept of race conditions, critical-sections, mutual exclusion, Solution to critical section problem, disabling interrupt, Peterson's solution, Sleep & Wake up, The Producer Consumer Problem

Unit-6 : Semaphores **57 – 67**

Semaphores, Wait and Signal, Binary Semaphores, Application of semaphores

Unit-7 : Scheduling **68 – 81**

Basic concepts, primitive and non-primitive scheduling, Types of scheduling - batch, interactive and real-time, Scheduling criteria, Scheduling algorithms, first come first serve (FCFS), shortest job first (SJF) and round robin (RR) scheduling

Unit-8 : Deadlocks **82 – 92**

What is deadlock, Conditions for deadlock, Deadlock detection, Resource allocation graph, Methods for handling deadlocks-ignore problem, recovery & prevention, deadlock avoidance (banker's algorithm), Congestion versus Deadlock

UNIT 1: INTRODUCTION TO OPERATING SYSTEM

UNIT STRUCTURE

1.1 Learning Objectives
1.2 Introduction
1.3 Definition of Operating System
1.4 OS as an Interface
1.5 OS as Resource Manager
1.6 Evolution and Types of Operating System
    1.6.1 Serial Processing
    1.6.2 Simple Batch Systems
    1.6.3 Multiprogrammed Batch System
    1.6.4 Time Sharing System
    1.6.5 Personal Computer Operating System
    1.6.6 Real-Time Operating System
    1.6.7 Distributed Operating System
1.7 Let Us Sum Up
1.8 Further Reading
1.9 Answers To Check Your Progress
1.10 Model Questions

1.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- interpret the basic functions of operating systems
- learn about serial processing, batch processing, multiprogramming
- describe time sharing system, real-time system
- describe personal computer system and distributed operating system

1.2 INTRODUCTION

There are many important reasons for studying operating systems (OS). It helps the users to understand the inner workings of a computer system. Operating system acts as an interface between users and computer system.

User interacts with the computer through operating system in order to accomplish their task.

The introductory concepts and principles of operating system will be the main issues for the discussion in this unit. Evolution of operating system along with its types will also be covered in this unit. In the next few units we will explore the concepts of process and treads.

1.3 DEFINITION OF OPERATING SYSTEM

Operating system is a set of programs that manages the computer hardware. It also provides a basis for application programs and acts as an intermediary between the user and the hardware. An amazing aspect of operating systems is how varied they are in accomplishing these tasks. Mainframe operating systems are designed primarily to optimize utilization of hardware. Personal computer (PC) operating systems support complex games, business applications, and everything in between. Operating systems for handheld computers are designed to provide an environment in which a user can easily interact with the computer to execute programs.

An operating system (OS) is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware. Any OS has three objectives:

- **Convenience:** An OS makes the computer more convenient to use.
- **Efficiency:** An OS allows the computer system resources to be used in an efficient manner.
- **Ability to evolve:** An OS should be constructed in such a way as to permit the effective development, testing, and introduction of new system functions without interfering with service.

1.4 OS AS AN INTERFACE

Operating system typically provides services in the following areas:

- **Program development:** The OS provides a variety of facilities and services, such as editors and debuggers, to assist the programmer in creating programs. Typically, these services are

in the form of utility programs that, while not strictly part of the core of the OS, are supplied with the OS and are referred to as application program development tools.

- **Program execution:** A number of steps need to be performed to execute a program. Instructions and data must be loaded into main memory, I/O devices and files must be initialized, and other resources must be prepared. The OS handles these scheduling duties for the user.

Access to I/O devices: Each I/O device requires its own peculiar set of instructions or control signals for operation. The OS provides a uniform interface that hides these details so that programmers can access such devices using simple reads and writes.

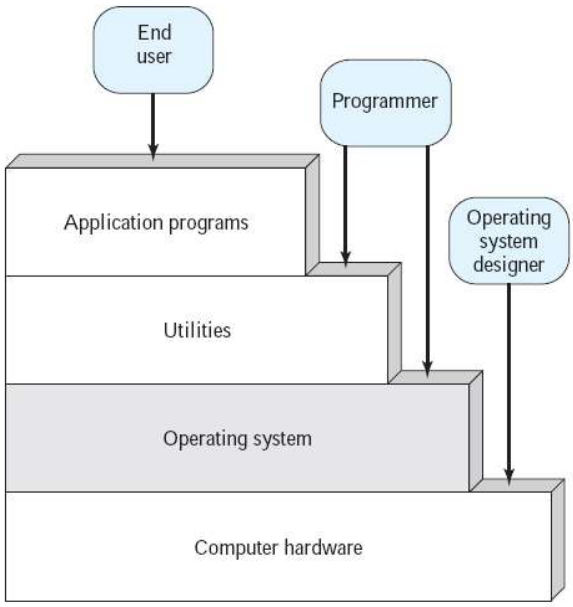


Fig 1.1: Layers and Views of a Computer System

- **Controlled access to files:** For file access, the OS must reflect a detailed understanding of not only the nature of the I/O device (disk drive, tape drive) but also the structure of the data contained in the files on the storage medium. In the case of a system with multiple users, the OS may provide protection mechanisms to control access to the files.

- **System access:** For shared or public systems, the OS controls access to the system as a whole and to specific system resources. The access function must provide protection of resources and data from unauthorized users and must resolve conflicts for resource contention.
- **Error detection and response:** A variety of errors can occur while a computer system is running. These include internal and external hardware errors, such as a memory error, or a device failure or malfunction; and various software errors, such as division by zero, attempt to access forbidden memory location, and inability of the OS to grant the request of an application. In each case, the OS must provide a response that clears the error condition with the least impact on running applications. The response may range from ending the program that caused the error to retrying the operation, to simply reporting the error to the application.
- **Accounting:** A good OS will collect usage statistics for various resources and monitor performance parameters such as response time. On any system, this information is useful in anticipating the need for future enhancements and in tuning the system to improve performance. On a multi-user system, the information can be used for billing purposes.

1.5 OS AS RESOURCE MANAGER

A computer is a set of resources for the movement, storage, and processing of data. For the control of these functions the OS is responsible for managing these resources.

When a computer (or network) has multiple users, the need for managing and protecting the memory, I/O devices, and other resources is even greater, since the users might otherwise interfere with one another. In addition, users often need to share not only hardware, but information (files, databases, etc.) as well. In short, this view of the operating system holds that its primary task is to keep track of who is using which resource, to grant resource requests, to account for usage, and to mediate conflicting requests from different programs and users.

Resource management includes multiplexing (sharing) resources in two ways: (a) in time and (b) in space. When a resource is time multiplexed, different programs or users take turns using it. First, one of the process gets to use the resource, then another, and so on. The other kind of multiplexing is space multiplexing where instead of the customers taking turns, each process gets a part of the resource.

1.6 THE EVOLUTION AND TYPES OF OPERATING SYSTEM

In attempting to understand the key requirements for an OS and the significance of the major features of a contemporary OS, it is useful to consider how operating systems have evolved over the years. Let us discuss it below :

1.6.1 Serial Processing

With the earliest computers, from the late 1940s to the mid-1950s, the programmer interacted directly with the computer hardware; there was no OS. These computers were run from a console consisting of display lights, toggle switches, some form of input device, and a printer. Programs in machine code were loaded via the input device (e.g., a card reader). If an error halted the program, the error condition was indicated by the lights. If the program proceeded to a normal completion, the output appeared on the printer. These early systems presented two main problems:

- **Scheduling:** Most installations used a hard copy sign-up sheet to reserve computer time. Typically, a user could sign up for a block of time in multiples of half hour or so. A user might sign up for an hour and finish in 45 minutes; this would result in wasted computer processing time. On the other hand, the user might run into problems, not finish in the allotted time, and be forced to stop before resolving the problem.

- **Setup time:** A single program, called a **job**, could involve loading the compiler plus the high-level language program (source program) into memory, saving the compiled program (object program) and then loading and linking together the object program and common functions. Each of these steps could involve mounting or dismounting tapes or setting up card decks. If an error occurred, the user typically had to go back to the beginning of the setup sequence. Thus, a considerable amount of time was spent just in setting up the program to run.

This mode of operation could be termed *serial processing*, reflecting the fact that users have access to the computer in series. Over time, various system software tools were developed in order to make serial processing more efficient. These include libraries of common functions, linkers, loaders, debuggers, and I/O driver routines that were available as common software for all users.

1.6.2 Simple Batch System

Early computers were very expensive, and therefore it was important to maximize processor utilization. The wasted time due to scheduling and setup time was unacceptable.

To improve utilization, the concept of a batch operating system was developed. It appears that the first batch operating system (and the first OS of any kind) was developed in the mid-1950s by General Motors for use on an IBM 701. The concept was subsequently refined and implemented on the IBM 704 by a number of IBM customers. By the early 1960s, a number of vendors had developed batch operating systems for their computer systems. IBSYS, the IBM operating system for the 7090/7094 computers, is particularly notable because of its widespread influence on other systems.

The central idea behind the simple batch-processing scheme is the

use of a piece of software known as the **monitor**. With this type of OS, the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor. Each program is constructed to branch back to the monitor when it completes processing, at which point the monitor automatically begins loading the next program.

1.6.3 Multiprogrammed Batch System

Even with the automatic job sequencing provided by a simple batch operating system, the processor is often idle. The problem is that I/O devices are slow when compared to the processor. This inefficiency can be handled.

We know that there must be enough memory to hold the OS (resident monitor) and one user program. Suppose that there is room for the OS and two user programs. When one job needs to wait for I/O, the processor can switch to the other job, which is likely not waiting for I/O. Furthermore, we might expand memory to hold three, four, or more programs and switch among all of them. This approach is known as **multiprogramming**, or **multitasking**. It is the central theme of modern operating systems.

Table 1.1 Effects of Multiprogramming on Resource Utilization

	Uniprogramming	**Multiprogramming**
Processor use	20%	40%
Memory use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

1.6.4 Time Sharing System

With the use of multiprogramming, batch processing can be quite efficient. However, for many jobs, it is desirable to provide a mode in which the user interacts directly with the computer. Indeed, for some jobs, such as transaction processing, an interactive mode is essential.

Today, the requirement for an interactive computing facility can be, and often is, met by the use of a dedicated personal computer or workstation. That option was not available in the 1960s, when most computers were big and costly. Instead, time sharing concept was developed. Just as multiprogramming allows the processor to handle multiple batch jobs at a time, multiprogramming can also be used to handle multiple interactive jobs. In this latter case, the technique is referred to as ***time sharing***, because processor time is shared among multiple users. In a ***time-sharing system***, multiple users simultaneously access the system through terminals, with the OS interleaving the execution of each user program in a short burst or quantum of computation.

Thus, if there are *n* users actively requesting service at one time, each user will only see on the average 1/*n* of the effective computer capacity, not counting OS overhead. However, given the relatively slow human reaction time, the response time on a properly designed system should be similar to that on a dedicated computer. Both batch processing and time sharing use multiprogramming.

1.6.5 Personal Computer Operating System

The next category is the personal computer operating system, where the job is to provide a good interface to a single user. They are widely used for word processing, spreadsheets, and Internet access. Common examples are Windows 98, Windows 2000, the Macintosh operating system, and Linux. Personal computer operating systems are so widely known that probably little introduction is needed. In fact, many people are not even aware that other kinds exist.

1.6.6 Real-Time Operating System

Another type of operating system is the real-time system. These systems are characterized by having time as a key parameter. For example in industrial process control systems, real-time computers have to collect data about the production process and use it to control machines in the factory. Often there are hard deadlines that must be met. For example, if a car is moving down an assembly line, certain actions must take place at certain instants of time, if a welding robot welds too early or too late, the car will be ruined. If the action absolutely *must* occur at a certain moment (or within a certain range), we have a **hard real-time system.**

Another kind of real-time system is a **soft real-time** system, in which missing an occasional deadline is acceptable. Digital audio or multimedia systems fall in this category. VxWorks and QNX are well-known real-time operating systems.

1.6.7 Distributed Operating System

The key concept of distributed operating system is transparency. It runs on multiple independent CPUs but appears as if it is centralized i.e., the use of multiprocessors are invisible to the user. In a true distributed system, users are not aware of where their programs are being run or where their files are residing. They should all be handled automatically and efficiently by the operating system. Distributed operating system often allow programs to run on several processors at the same time, thus requiring more complex processor *scheduling algorithms* in order to achieve maximum utilization of CPU's time.

Distributed systems are considered to be more reliable than system having single processor. They perform even if certain part of the hardware is malfunctioning.



CHECK YOUR PROGRESS

Q1. Fill in the blanks:

(a) An _____ is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware.

(b) In a batch-processing scheme, the software used is known as the _____.

(c) In a _____ system, multiple users can simultaneously access the system through terminals.

(d) In _____ system, the user submits the job on cards or tapes.

(e) In _____, processor time is shared among multiple users.

(f) The _____ operating system perform even if certain part of the hardware is malfunctioning



1.7 LET US SUM UP

- An ***Operating System*** is a program that controls the execution of application programs and acts as an interface between applications and the computer hardware
- A computer is a set of resources for the movement, storage, and processing of data and for the control of these functions the OS is responsible for managing these resources.
- ***Serial Processing***, reflecting the fact that users have access to the computer in series.
- In a Batch-processing scheme, a software is used known as the ***monitor***.
- In ***Batch Processing system***, the user no longer has direct access to the processor. Instead, the user submits the job on cards or tape to a computer operator, who batches the jobs together sequentially and places the entire batch on an input device, for use by the monitor.
- In a ***time-sharing system***, multiple users can simultaneously access the system through terminals.



1.8 FURTHER READING

1. Tanenbaum, A. S., & Woodhull, A. S. (1987). Operating systems: design and implementation (Vol. 2). Englewood Cliffs, NJ: Prentice-Hall.
2. Deitel, H. M. (2004). Operating systems. Pearson Education India.



1.9 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1. (a) operating system

(b) monitor

(c) time sharing

(d) batch processing

(e) time sharing

(f) distributed



1.10 MODEL QUESTIONS

Q1. What is an operating system ?
Q2. Write briefly about the functions of an operating system ?
Q3. How does operating system act as an interface?
Q4. How does operating system act as a resource manager?
Q5. What is serial processing?
Q6. What is a batch system?
Q7. What is multiprogramming or multitasking?
Q8. What is time sharing system?
Q9. What is real time operating system?

*****

UNIT 2: PROCESSES

UNIT STRUCTURE

2.1 Learning Objectives
2.2 Introduction
2.3 Process
    2.3.1 Process Control Block
2.4 Process States
    2.4.1 Creation of Processes
    2.4.2 Termination of Processes
2.5 Let Us Sum Up
2.6 Further Reading
2.7 Answers To Check Your Progress
2.8 Model Questions

2.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- define process
- describe process control block
- illustrate the different states of processes
- learn about process model
- learn how processes can be created and how they can be terminated
- learn about the concept of thread and its usage

2.2 INTRODUCTION

In the earlier unit, we have learnt about the concept of operating systems. Different operating systems like batch, multiprogramming, time sharing, real time and distributed operating systems were discussed in the previous unit. Initially, computer could run only one program at a time. This program had complete control of the system and had access to all the system's resources. In contrast, the current-day computer systems allow multiple programs to

be loaded into memory and executed concurrently. Processor management is concerned with the management of processors (CPUs) i.e., the allocation of processes to a processor. A *process* is basically a program while it is being executed i.e., it is a running program with some specific tasks to do.

A system consists of a collection of processes: *operating system processes* executing system code and *user processes* executing user code. All these processes can execute concurrently with the CPU or CPUs multiplexed among them. By switching the CPU between processes, the operating system can make the computer more productive.

In this unit, we will discussing the concept of process in details which includes process control blocks, process model, creation and termination of processes, etc. In the next unit, we will explore the concepts of threads.

2.3 PROCESS

A process is basically a program while it is being executed. A process can be defined as:

- a program in execution
- an instance of a program running on a computer
- the entity that can be assigned to and executed on a processor
- a unit of activity characterized by the execution of a sequence of instructions, a current state, and an associated set of system resources

We can also think of a process as an entity that consists of a number of elements. Two essential elements of a process are **program code** (which may be shared with other processes that are executing the same program) and a **set of data** associated with that code. Let us assume that the processor begins to execute a program code, and we refer to this executing entity as a process.

2.3.1 Process Control Block

The OS groups all information that it needs about a particular process

into a data structure called a **process control block** (PCB). Fig: 2.1 depicts a PCB.

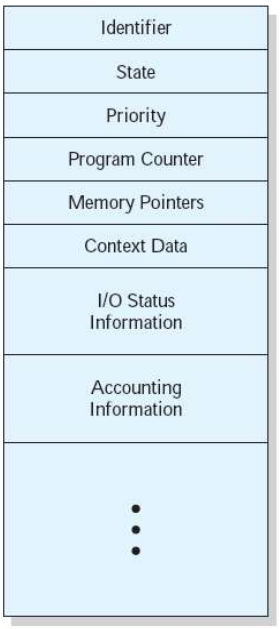


Fig 2.1: Process Control Block

At any given point in time, *during execution*, any process can be uniquely characterized by a number of elements, including the following:

- **Identifier:** A unique identifier associated with the process, to distinguish it from all other processes.
- **State:** If the process is currently executing, it is in the running state.
- **Priority:** Priority level relative to other processes.
- **Program counter:** The address of the next instruction in the program to be executed.
- **Memory pointers:** Includes pointers to the program code and data associated with this process, plus any memory blocks shared with other processes.
- **Context data:** These are data that are present in registers in the processor while the process is executing.
- **I/O status information:** Includes outstanding I/O requests, I/O devices (e.g., tape drives) assigned to this process, a list of files in use by the process, and so on.

- **Accounting information:** May include the amount of processor time, clock time used, time limits, account numbers, and so on.

The significant point about the Process Control Block (PCB) is that it contains sufficient information so that it is possible to interrupt a running process and later resume execution as if the interruption had not occurred. The process control block is the key tool that enables the OS to support multiple processes and to provide multiprocessing. When a process is interrupted, the current values of the program counter and the processor registers (context data) are saved in the appropriate fields of the corresponding process control block and the state of the process is changed to some other value, such as *blocked* or *ready*. The OS is now free to put some other process in the running state. The program counter and context data for this process are loaded into the processor registers and this process now begins to execute.

Thus, we can say that a process consists of program code and associated data along with a process control block. For a single-processor computer, at any given time, at most one process is executing and that process is in the *running* state.

2.4 PROCESS STATES

Although each process is an independent entity, with its own program counter and internal state, processes often need to interact with other processes. One process may generate some output that another process uses as input.

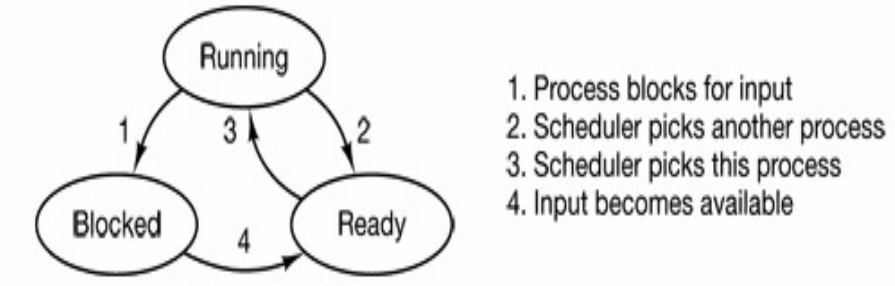


Fig 2.2: Process transition diagram

In Fig. 2.2, we see a state transition diagram showing the three states of a process.

- ***Running*** (using the CPU at that instant).
- ***Ready*** (runnable; temporarily stopped to let another process run).
- ***Blocked*** (unable to run until some external event happens)

Logically, the first two states are similar. In both cases, the process is willing to run, but only in the second one, there is temporarily no CPU available for it. The third state is different from the first two in that the process cannot run, even if the CPU is free.

There are four transitions that are possible among these three states, as shown. ***Transition 1*** occurs when a process discovers that it cannot continue.

Transitions 2 and 3 are caused by the process scheduler, a part of the operating system. ***Transition 2*** occurs when the scheduler interrupt the running process, and it is time to let another process to have some CPU time. ***Transition 3*** occurs when all the other processes have had their fair share and it is time for the first process to get the CPU to run again.

Transition 4 occurs when the external event for which a process was waiting (such as the arrival of some input) happens. If no other process is running at that instant, transition 3 will be triggered and the process will start running. Otherwise it may have to wait in *ready* state for a little while until the CPU is available and its turn comes.

Using the process model, it becomes much easier to think about what is going on inside the system. Some of the processes run programs that carry out commands typed in by a user. Other processes are part of the system and handle tasks such as carrying out requests for file services or managing the details of running a disk or a tape drive. When a disk interrupt occurs, the system makes a decision to stop running the current process and run the disk process, which was blocked waiting for that interrupt. Thus, instead of thinking about interrupts, we can think about user processes, disk processes, terminal processes, and so on, which block when they are waiting for something to happen. When the disk has been read or the character typed, the process waiting for it is unblocked and is eligible to run again.

2.4.1 Creation of Processes

When a new process is to be added, then the OS builds data structures, that are used to manage the process and allocates address space in main memory to the process. We have already described these data structures while describing PCB.

New batch job	The OS is provided with a batch job control stream, usually on tape or disk. When the OS is prepared to take on new work, it will read the next sequence of job control commands.
Interactive logon	A user at a terminal logs on to the system.
Created by OS to provide a service	The OS can create a process to perform a function on behalf of a user program, without the user having to wait (e.g., a process to control printing).
Spawned by existing process	For purposes of modularity or to exploit parallelism, a user program can dictate the creation of a number of processes.

Fig 2.3: Events for creation of a process

Four common events lead to the creation of a process, as indicated in Fig. 2.3. In a batch environment, a process is created in response to the submission of a job. In an interactive environment, a process is created when a new user attempts to log on. In both cases, the OS is responsible for the creation of the new process. An OS may also create a process on behalf of an application. For example, if a user requests that a file be printed, the OS can create a process that will manage the printing. The requesting process can thus proceed independently of the time required to complete the printing task.

Traditionally, the OS created all processes in a way that was transparent to the user or application program, and this is still commonly found with many contemporary operating systems. However, it can be useful to allow one process to cause the creation of another. For example, an application process may generate another process to receive data that the application is generating and to organize those data into a form suitable for later analysis. The new process runs in parallel to the original process and is activated from time to time when new data are

available. This arrangement can be very useful in structuring the application. As another example, a server process (e.g., print server, file server) may generate a new process for each request that it handles. When the OS creates a process at the explicit request of another process, the action is referred to as **process spawning**.

When one process spawns another, the former is referred to as the **parent process**, and the spawned process is referred to as the **child process**. Typically, the "related" processes need to communicate and cooperate with each other.

In Fig 2.3, we have discussed the events that lead to the creation of a new process. Having discussed the data structures associated with a process, we are now in a position to describe briefly the steps involved in actually creating the process. Once the OS decides to create a new process, it can proceed as follows:

1. Assign a unique process identifier to the new process: A new entry is added to the primary process table, which contains one entry per process.

2. Allocate space for the process: This includes all elements of the process image. Thus, the OS must know how much space is needed for the private user address space (programs and data) and the user stack. These values can be assigned by default based on the type of process, or they can be set based on user request at job creation time. If a process is spawned by another process, the parent process can pass the needed values to the OS as part of the process-creation request. If any existing address space is to be shared by this new process, the appropriate linkages must be set up. Finally, space for a process control block must be allocated.

3. Initialize the process control block. The process identification portion contains the ID of the process plus other appropriate IDs, such as the ID of the parent process. The processor state information is

initialized with most entries as zero, except for the program counter (set to the program entry point) and system stack pointers (set to define the process stack boundaries).The process control information is initialized based on standard default values plus attributes that have been requested for this process. For example, the process state would typically be initialized to Ready or Ready/Suspend. The priority may be set by default to the lowest priority unless an explicit request is made for a higher priority. Initially, the process may own no resources (I/O devices, files) unless there is an explicit request for these or unless they are inherited from the parent.

4. Set the appropriate linkages: For example, if the OS maintains each scheduling queue as a linked list, then the new process must be put in the Ready or Ready/Suspend list.

5. Create or expand other data structures: For example, the OS may maintain an accounting file on each process to be used subsequently for billing or for performance assessment purposes.

2.4.2 Termination of Process

A running process ends its life when it has completed executing its last statement. The process may also terminate due to one of the following conditions:

1. Normal exit (voluntary).
2. Error exit (voluntary).
3. Fatal error (involuntary).
4. Killed by another process (involuntary).

Most processes terminate because they have finished their work. When a compiler has compiled the program given to it, the compiler executes a system call to tell the operating system that it is finished. This call is exit in UNIX and ExitProcess in Windows. Screen-oriented programs also support voluntary termination. Word processors, internet browsers and similar programs always have an icon or menu item that the user can click to tell the process to remove any temporary

files it has opened and then terminated.

The second reason for termination is that the process discovers a fatal error. For example, if a user types the command **cc foo.c** to compile the program *foo.c,* and no such file exists, the compiler simply exits. Screen-oriented interactive processes generally do not exit when given bad parameters. Instead, they pop up a dialog box and ask the user to try again.

Normal completion	The process executes an OS service call to indicate that it has completed running.
Time limit exceeded	The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.
Memory unavailable	The process requires more memory than the system can provide.
Bounds violation	The process tries to access a memory location that it is not allowed to access.
Protection error	The process attempts to use a resource such as a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.
Arithmetic error	The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.
Time overrun	The process has waited longer than a specified maximum for a certain event to occur.
I/O failure	An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).
Invalid instruction	The process attempts to execute a nonexistent instruction (often a result of branching into a data area and attempting to execute the data).
Privileged instruction	The process attempts to use an instruction reserved for the operating system.
Data misuse	A piece of data is of the wrong type or is not initialized.
Operator or OS intervention	For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).
Parent termination	When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.
Parent request	A parent process typically has the authority to terminate any of its offspring.

Fig 2.4: Some possible reasons for process termination

The third reason for termination is an error caused by the process, often due to a program bug. Examples include executing an illegal instruction, referencing nonexistent memory, or dividing by zero. In some systems (e.g. UNIX), a process can tell the operating system that it wishes to handle certain errors itself, in which case the process is signaled (interrupted) instead of being terminated when one of the errors occurs.

The fourth reason why a process might terminate is that a process executes a system call telling the operating system to kill some other

process. In UNIX this call is called kill. The corresponding Windows function is TerminateProcess. In both cases, the killer must have the necessary authorization to do in the killee. In some systems, when a process terminates, either voluntarily or otherwise, all processes it created are immediately killed as well. However, neither UNIX nor Windows works this way.



CHECK YOUR PROGRESS

Q1. Fill in the blanks:

(a) A program in execution can be referred to as a _____.

(b) Two essential elements of a process are _____ and _____

(c) Processes can be distinguished from all other processes by an ___

(d) If the process is currently executing, it is in the _____ state.

(e) A process can have three states named _____, _____ and _____.

(f) When the operating system creates a process at the explicit request of another process, the action is referred to as _____

(g) When one process creates another process, the former is referred to as the _____, and the spawned process is referred to as the _____.



2.5 LET US SUM UP

- The most fundamental concept in a modern OS is the ***process***. The principal function of the OS is to create, manage, and terminate processes.
- To perform process management functions, the OS maintains a description of each process, or process image, which includes the address space within which the process executes, and a ***process control block***.

- During its lifetime, a process moves among a number of states. The most important of these are ***Ready***, ***Running***, and ***Blocked***.
- A ***ready process*** is one that is not currently executing but that is ready to execute.
- The ***running process*** is that process that is currently being executed by the processor. In a multiple-processor system, more than one process can be in this state
- A ***blocked process*** is waiting for the completion of some event, such as an I/O operation.
- Operating systems must provide a mechanism for ***parent processes*** to create new ***child processes***. The parent may wait for its children to terminate before proceeding, or the parent and children may execute concurrently.



2.6 FURTHER READING

1. Tanenbaum, A. S., & Woodhull, A. S. (1987). Operating systems: design and implementation (Vol. 2). Englewood Cliffs, NJ: Prentice-Hall.
2. Deitel, H. M. (2004). Operating systems. Pearson Education India.



2.7 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1. (a) process

(b) program code, set of data associated with the code

(c) identifier

(d) running

(e) running, ready, blocked

(f) process spawning

(g) parent, child

(h) lightweight

(i) Multithreading



2.8 MODEL QUESTIONS

Q1. What is a process? How can a process can be uniquely identified?

Q2. What are the different states a process can have?

Q3. Explain briefly the process model with a diagram?

Q4. How can a process be created? What is parent and child process?

Q5. How can a process can be terminated?

*****

UNIT 3 : THREADS

UNIT STRUCTURE

3.1 Learning Objectives
3.2 Introduction
3.3 Threads
3.4 Uses of Thread
3.5 Design Issues of Thread
3.6 Thread Management
3.7 Let Us Sum Up
3.8 Further Reading
3.9 Answers to Check Your Progress
3.10 Model Questions

3.1 LEARNING OBJECTIVES

After going through this unit, you would be able to:

- define thread
- describe the different states of thread
- describe the different advantages of using thread
- describe the design issues of thread

3.2 INTRODUCTION

In the previous units, we have learned about the different types of operating system like real time, multiprogramming and distributed operating system. We have also learnt about the concept of process. We have learnt about the creation and termination of process along with the different states in the life cycle of a process.

In this unit, we will learn about the concept of threads. The concept of

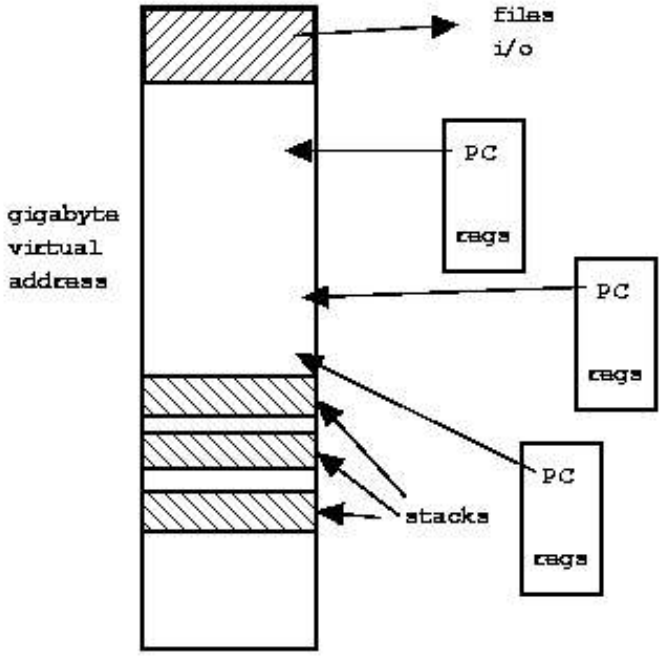
multiprogramming is used for many applications to achieve efficiency by forking multiple processes and distributing its work against them. But every process consumes memory and sharing information across processes is not easy so it is needed to share memory corresponding to the code and data in the memory of the process. To achieve this, the concept of threads is introduced and hence it is also termed as a light-weight process. By default, every process has a thread. In the next unit we will learn about interprocess communication in detail.

3.3 THREADS

A thread is a ***lightweight process***. The thread has a program counter that keeps track of which instruction to execute next. It has registers, which hold its current working variables. It has a stack, which contains the execution history, with one frame for each procedure called but not yet returned from. Although a thread must execute in some process, the thread and its processes are different concepts and can be treated separately. Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.

Threads are multiple executing entities like processes and are used to effectively utilize system resources while introducing the concept of multiprogramming or concurrency. The smallest unit of processing that can be scheduled by an operating system is termed as thread. In a process, all threads share address space, global variables, child processes, pending alarms and open files. Every thread has its own program counter, registers, stack and state. Threads execute within a process.

Each thread has its own stack, PC, registers. Threads of the same process share code, data and resources with one another. Figure 3.1 shows a thread.



- **Life cycle of a Thread:** Since a thread is analogous to a process thread states and thread state transitions are analogous to process states and process state transitions.

  A thread can be in any one of the four states: ***New, Runnable, Blocked*** or ***Dead***. In its life span a thread can move between its different states. The ***block*** diagram of life cycle of a thread is shown in figure 3.2.

  - **New :** A new thread program is initiated once it is created at this state, but before the start () method has been called. At this moment as no CPU cycles are being consumed the thread is inactive.
  - **Runnable :** In runnable state a thread is either currently running or ready to run. In multithreading different threads are allocated CPU time one at a time, hence a thread may be in a runnable state.
  - **Block/Waiting :** When a thread issues an input/output request it enters the blocked state. A thread is blocked from executing by the operating system until the I/O for which the thread is waiting is completed by the operating system. Then the thread returns to the started state to resume its execution. When a thread is selected to run, a thread scheduling algorithm is

employed by the operating system, to determine which thread will be the next to run.

- **Dead:** A thread is in the dead state if it has completed the code in its run() method or if it terminates abnormally due to an exception. At this state there is no further consumption of CPU cycles.

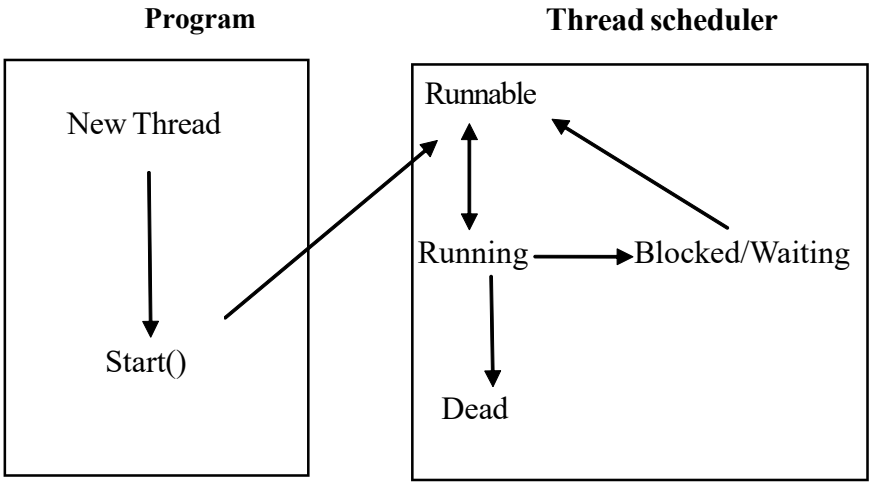


Fig.3.2. Different stages of a thread

The thread scheduler picks one of the thread from the runnable thread pool and changes its state to Running. Then the CPU starts executing this thread. A thread can change the state to Runnable, Dead or Blocked from running state and depends on time slicing, thread completion of run() method or waiting for some resources.

- **Similarities with process**

1. Like processes threads share CPU and only one thread is active (running) at a time.
2. Like processes, threads within processes execute sequentially.
3. Like processes, thread can create children.
4. Like process, if one thread is blocked, another thread can run.

- **Differences with process**

1. Unlike processes, threads are not independent of one another.
2. Unlike processes, all threads can access every address in the task.
3. Unlike processes, threads are designed to assist one other. It is to be noted that processes might or might not assist one another because processes may originate from different users.

3.4 USES OF THREAD

In many applications multiple activities are going on at once and performing these multiple processes have many overheads. With threads, an application can avoid per-process overheads, as threads have the ability for the parallel entities to share an address space and all of its data among themselves. Using thread is more advantageous as they do not have any resources attached to them. Hence thread creation, deletion and switching is cheaper than processes. Moreover thread can easily share their address space. Threads can execute in parallel on multiprocessors.

Following are some reasons why we use threads in designing operating systems.

1. A process with multiple threads make a great server, for example printer server.
2. Because threads can share common data, they do not need to use interprocess communication.
3. Because of the very nature, threads can take advantage of multiprocessors.
4. Threads are cheap in the sense that–
   - ❍ They only need a stack and storage for registers; therefore, threads are cheap to create.
   - ❍ Threads use very little resources of the operating system in which they are working. That is, threads do not need new address space, global data, program code or operating system resources.
   - ❍ Context switching is fast when working with threads. The reason for this is that we only have to save or restore PC, SP and registers.

But the biggest drawback is that there is no protection between threads. Some other drawbacks are :

Blocking : The major disadvantage is that if the kernel is single threaded, a system call of one thread will block the whole process and CPU may be idle during the blocking period.

Security : Since there is an extensive sharing among threads there is a potential problem of security. It is quite possible that one thread overwrites the stack of another thread, (or damaged shared data). Although it is very unlikely since threads are meant to cooperate on a single task.

3.5 DESIGN ISSUES OF THREAD

Every process has at least one thread. Design issues of threads are those that enable multitasking facility. An application may have a single process or multiple processes and correspondingly there are two approaches single-threaded approach and multithreaded approach.

❍ **Different approaches:**

(i) Single Threaded approaches: In a single-threaded approach a process has a single execution path. By default, every process is singly threaded. In MS-DOS and some versions of UNIX this type of approach is supported. MS-DOS follows single process, single thread approach whereas UNIX supports multiple processes, single thread approach.

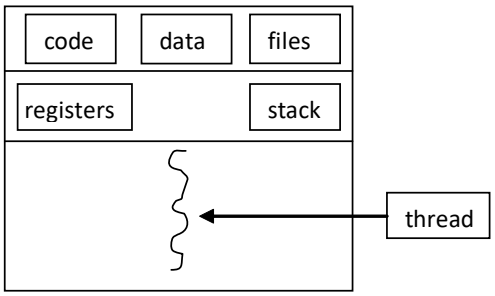


Fig. 3.3 (a). A single threaded approach

(ii) Multithreaded Approaches: In multithreaded approaches there are multiple threads as shown in figure 3.3 (b). A multithreading model describes how the user and the kernel threads map into each other. Multithreaded approach may be many-to-one, one-to-one or many-to-many.

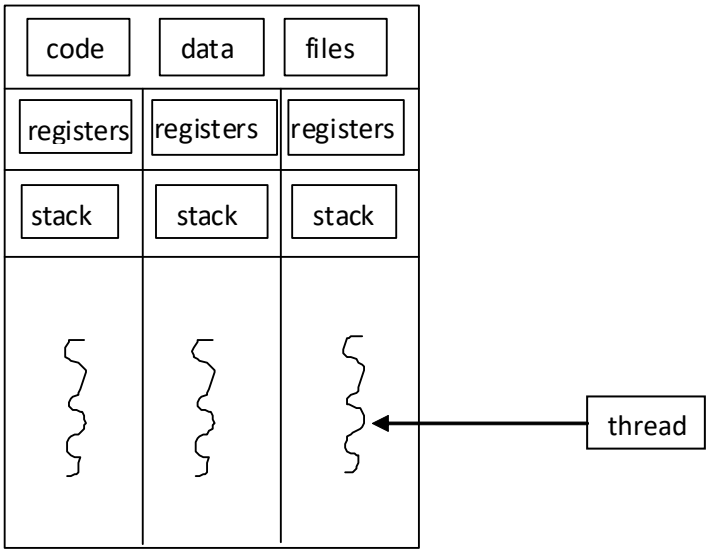


Fig. 3.3(b) Multithreaded approach

A multithreaded approach may have single process like OS/161 r multiple processes like modern UNIX (Linux, Solaris) or Windows.

a) **On-to- one :** In one-to-one approach each of the user-level thread maps to kernel thread. Examples are Windows 95/98/NT/2000 and Linux.
A thread contains a thread id, register set, separate user and kernel stacks and private data storage area. The stacks, register set and private storage area together are termed as context of the threads. Fig. 3.4 (a) shows one-to one mapping of user and kernel level thread.

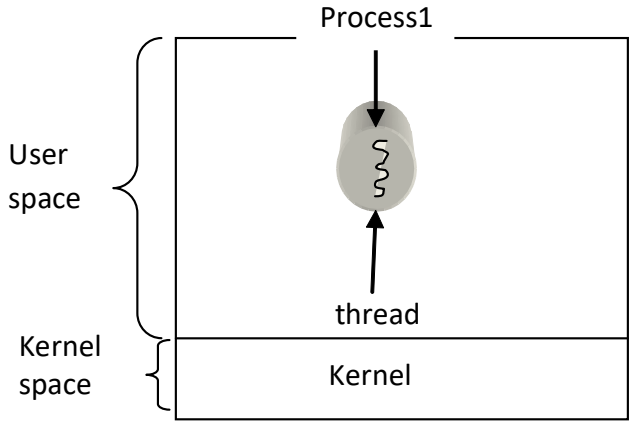


Fig. 3.4 (a) One-to-one mapping of kernel to user level thread

b) **Many-to one :** In thread model of many to one mapping of user to kernel level mapping, the mapping is shown in fig. 3.4(b).

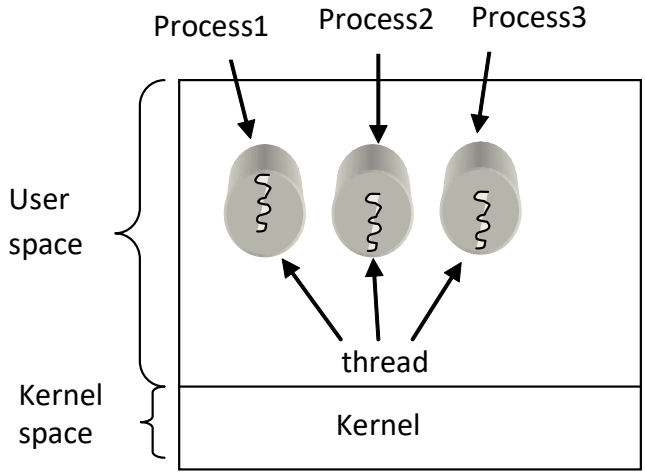


Fig. 3.4(b) Many-to-one mapping of user to kernel level thread



CHECK YOUR PROGRESS

State whether the following statements are true (T) or false (F).

1. Every process has a thread.
2. A new thread program is initiated after start () method is being called.
3. A thread is blocked when it issues an I/O request.
4. A thread may be in Dead state if it terminates abnormally due to an exception.
5. Threads within a process can't execute sequentially.
6. Threads do not share an address space and all of its data among themselves.
7. Threads can execute in parallel on multiprocessors.
8. Threads are cheap to create.
9. MSDOS follows multiple process, single thread approach.
10. Multithreaded approach may be many-to-one, one-to-one or many-to many.

3.6 Thread Management

Depending on how much the kernel and the application program know about the threads, it is managed in both kernel and user level and accordingly a thread may be of two types : (a) Kernel-space thread (KLT) and (b) User-space thread (ULT).

(a) Kernel-Level thread: It is implemented by the kernel. Creation, termination and status checking is performed by system calls. When a process makes a create_thread system call, the kernel assigns an id to it and a thread control block (TCB) is allocated. The TCB contains a pointer to the PCB of the process.

(b) User-Level Threads: It avoids the kernel and manages the tables itself. Also termed as cooperative multitasking where the task

defines a set of routines, switched in between by manipulating the stack pointer. The problem associated with it is that a single thread can monopolize the time slice thus starving the other threads within the task.



3.7 LET US SUM UP

- Threads are multiple executing entities like processes and are used to effectively utilize system resources while introducing the concept of multiprogramming or concurrency.
- Each thread has its own stack, PC, registers. Threads of the same process share code, data and resources with one another.
- A thread can be in any one of the four states: New, Runnable, Blocked or Dead. In its life span a thread can move between its different states.
- Design issues of threads are that to enable multitasking facility an application may have a single process or multiple processes and correspondingly there are two approaches-single-threaded approach and multithreaded approach.
- In a single-threaded approach, a process has a single execution path. By default every process is singly threaded.
- A multithreading model describes how do the user and kernel threads map into each other. Multithreaded approach may be many-to-one, one-to-one or many-to-many.
- In one-to-one approach each of the user-level thread maps to kernel thread. Examples are Windows 95/98/NT/2000 and Linux.
- Depending on how much the kernel and the application program know about the threads, it is managed in both kernel and user level and accordingly a thread may be of two types: User-space thread (ULT) and Kernel-space thread (KLT).



3.8 FURTHER READING

1. Dhamdhere, D. M. (2006). Operating Systems: A Concept-based Approach, 2E. Tata McGraw-Hill Education.
2. Silberschatz, A., Galvin, P. B., Gagne, G., & Silberschatz, A. (1998). Operating system concepts (Vol. 4). Reading: Addison-wesley.



3.9 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: Yes
Answer to Q2: No
Answer to Q3: Yes
Answer to Q4: Yes
Answer to Q5: No
Answer to Q6: No
Answer to Q7: Yes
Answer to Q8: Yes
Answer to Q9: No
Answer to Q10: Yes



3.10 MODEL QUESTIONS

Q1. Define thread.

Q2. What are the benefits of threads?

Q3. What are the different types of threads?

Q4. What are the possible thread states?

Q5. How do threads share information?

Q6. How are threads managed?

Q7. What are the similarities and differences between thread and process?

****

UNIT 4 : INTERPROCESS COMMUNICATION

UNIT STRUCTURE

4.1 Learning Objectives
4.2 Introduction
4.3 Interprocess Communication (IPC)
4.4 Shared Memory Model
4.5 Message Passing Model
    4.5.1 Types of Message Passing
4.6 Let Us Sum Up
4.7 Further Reading
4.8 Answers to Check Your Progress
4.9 Model Questions

4.1 LEARNING OBJECTIVES

After going through this unit you will be able to:

- define interprocess communication
- describe message passing model
- describe shared memory model

4.2 INTRODUCTION

In the previous unit, we had discussed the concept of operating system. Different types of operating system like real time, time sharing, distributed operating system etc were discussed in detail. We have also discussed the concept of processes and threads. Topics like creation of threads, termination of threads, process life cycle were discussed in detail. We have already learnt about different types of threads in addition to topics like multithreading. In this unit, we will explore how communication between processes works. Independent and cooperating process are also covered in this unit. In addition to this, different types of Interprocess communication methods like message

passing shared memory model are discussed in detail in this unit. In the next unit we will look into problems associated with process synchronization and process coordination.

4.3 INTERPROCESS COMMUNICATION (IPC)

We have already come across the concept of process in unit 2. In this unit, we will discuss how communication takes place between different processes. A process is a program in execution. Processes frequently need to communicate with other processes, preferably in a well-structured way not using interrupts. This is termed as Interprocess Communication. Interprocess communication (IPC) can also be defines as a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them.

Why do we need Interprocess Communication?

Some of the reasons why we may need interprocess communication between processes are listed below:

- **Single user, multiple processes :** A single user may request some activities which need the operating system to run different processes. These processes will need a way to communicate so that the user requests are performed smoothly.

- **Information sharing :** It may happen that many processes require the same set of data or information, so it will be easier to share that information instead of storing redundant information.

- **Modularity :** It is easier for the operating system to divide most of the work requested by the user into sub tasks where they can be parrallely computed. The operating system uses multiple processes, threads, functions etc to perform these sub tasks.

A process can be categorized into two types: independent process or cooperating process. A process is independent if it cannot affect or be affected by other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process. All processes executing in multiprogrammed operating system environment compete with other processes for system resources such as memory and CPU. Therefore they must be synchronized with each other while accessing.

Processes can communicate with each other using two methods:

- **Shared Memory Model :** In this mechanism, one process writes a message in the shared memory location and another process reads the message from that location.

- **Message Passing Model :** In this mechanism, one process sends a message to the other process and expects it to read the message.

Operating systems can choose any one of these two methods for communication between processes. Let us now look into these methods for interprocess communication in detail.

4.4 SHARED MEMORY MODEL

In shared memory model, a region of shared memory needs to be established. This shared region of memory resides within the address space of any process that creates the shared memory segment. Any other process that wants to communicate with this process must connect this shared memory region to its address space.

Let us look at an example for shared memory model. Suppose, there are two processes P1 and P2, both of which are executing simultaneously and share some data and information. Process P1 executes its program and generates some information which it stores in the shared memory region as a record. Process P2 needs to use this information for further processing.

So, process P2 will now check the shared memory region and upon finding the record stored by process P1 use it accordingly. A representation of this shared memory model is presented in figure 4.1.

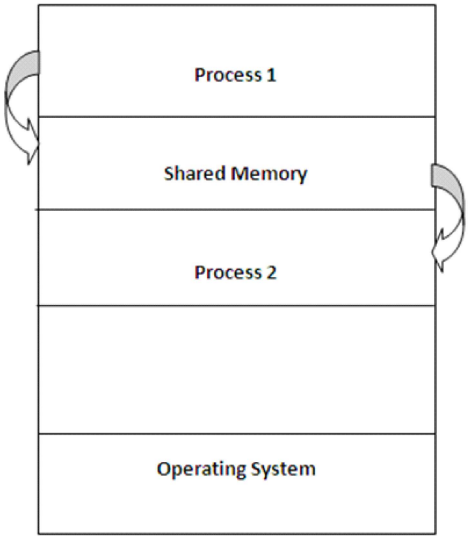


Fig 4.1: Shared Memory Model

The shared memory region can be used as a medium to both deliver information to other processes as well as to receive information from other processes. In figure 4.1, the process P1 is storing its information in the shared memory region. Process P2 on the other hand is accessing the information stored by the process P1 in the shared memory region. The operating system is also located in one part of the computer memory. Any other process that wants to communicate with the processes need to connect to the shared memory region in the memory.

4.5 MESSAGE PASSING MODEL

Message passing is one of the common mechanisms of communication between processes. In the message-passing mechanism, communication takes place by way of messages exchanged among the cooperating processes. A communication link has to be established between the processes before message passing can begin. Once the communication link has been established, message passing can begin.

There are two primitives that are used for message passing. They are described below:

- Send (Message, Destination) or Send (Message)
- Receive (Message, Source) or Receive (Message)

The message structure generally consists of a header part and a body part. The header part is used to store information like the message type, destination address, source address, length of the message etc. The body part contains the message itself. The message can be of fixed size or variable size. Keeping the message length fixed reduces the overhead of the operating system designer. Whereas, keeping the message size variable makes the operating system design complex.

Figure 4.2 shows an example for message passing mechanism. In this example, process P1 sends a message to P2 using the primitive send (message). This message is received by the process P2 using the primitive receive (message). The communication link between the processes needs to be established first.

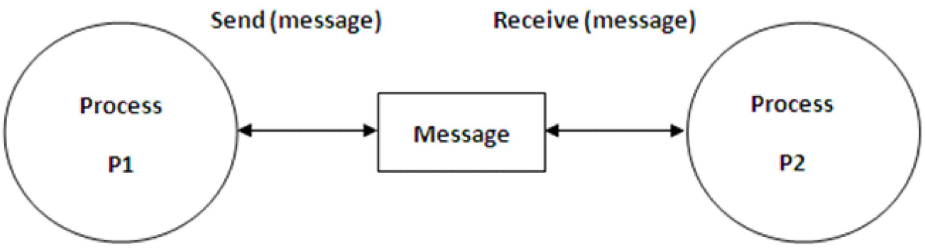


Fig 4.2: Message Passing Model

4.5.1 Types of Message Passing

Message passing mechanism can be classified into two categories. They are:

- Synchronous message passing
- Asynchronous message passing

In synchronous message passing, both the message sender process and message receiver process need to be in running or execution mode. Both processes are required to wait to for the other process while sending and receiving the messages.

In asynchronous message passing mechanism however, the processes need not be in running mode. In this mode, the receiver process may be busy performing other activities or may even not be in running mode.

The synchronous mechanism is simpler to design than asynchronous message passing mechanism. So, in object oriented programming languages like Java and Smalltalk, synchronous message passing mechanism is used. Asynchronous message passing works even for processes that are not running so this mechanism needs storage capabilities in order to store the messages or to retransmit the messages. The storage is needed so that the processes can read the messages when it is back to running mode.



CHECK YOUR PROGRESS

Q1. Define IPC.

Q2. What are independent processes and cooperating processes?

Q3. Differentiate between message passing and shared memory mechanisms.

Q4. Define synchronous message passing model.



4.6 LET US SUM UP

- Processes frequently need to communicate with other processes, preferably in a well-structured way not using interrupts. This is termed as Interprocess Communication.
- In shared memory model, a region of shared memory needs to be established.
- The shared region of memory resides within the address space of any process that creates the shared memory segment.

- In the message-passing mechanism, communication takes place by way of messages exchanged among the cooperating processes.
- A communication link has to be established between the processes before message passing can begin.
- In synchronous message passing, both the message sender process and message receiver process need to be in running or execution mode.
- In asynchronous message passing mechanism however, the processes need not be in running mode. In this mode, the receiver process may be busy performing other activities or may even not be in running mode.



4.7 FURTHER READING

1. Tanenbaum, A. S., & Woodhull, A. S. (1987). Operating systems: design and implementation (Vol. 2). Englewood Cliffs, NJ: Prentice-Hall.

2. Silberschatz, A., Galvin, P. B., Gagne, G., & Silberschatz, A. (1998). Operating system concepts (Vol. 4). Reading: Addison-Wesley.



4.8 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: Interprocess communication (IPC) can also be defines as a mechanism which allows processes to communicate each other and synchronize their actions. The communication between these processes can be seen as a method of co-operation between them.

Answer to Q2: A process is independent if it cannot affect or be affected by other processes executing in the system. Any process that does not share data with any other process is independent. A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

Answer to Q3: In shared memory mechanism, for interprocess communication, one process writes a message in the shared memory location and another process reads the message from that location. Whereas, in message passing model one process sends a message to the other process and expects it to read the message for interprocess communication.

Answer to Q4 : In synchronous message passing, both the message sender process and message receiver process need to be in running or execution mode. Both processes are required to wait to for the other process while sending and receiving the messages.



4.9 MODEL QUESTIONS

Q1. Define inter process communication.
Q2. Why do we need inter process communication?
Q3. What are the different methods that can be used for inter process communication?
Q4. Explain shared memory model using an example.
Q5. Explain message passing model using an example.
Q6. What are the advantages of message passing model over shared memory model?
Q7. Give the disadvantages of message passing model over shared memory model?
Q8. Differentiate between synchronous and asynchronous message passing.

****

UNIT 5 : PROCESS SYNCHRONIZATION AND CO-ORDINATION

UNIT STRUCTURE

5.1 Learning Objectives

5.2 Introduction

5.3 Process Synchronization and Co-ordination

- 5.3.1 Race Condition
- 5.3.2 Critical Sections

5.4 Mutual Exclusion

- 5.4.1 Disabling Interrupts
- 5.4.2 Peterson's Solutions
- 5.4.3 Sleep and Wake Up
- 5.4.4 Semaphores

5.5 Let Us Sum Up

5.6 Further Reading

5.7 Answers to Check Your Progress

5.8 Model Questions

5.1 LEARNING OBJECTIVES

After going through this unit you will be able to :

- describe how processes co-ordinate with each other
- describe race condition, critical section problems
- describe solutions to mutual exclusion problem.

5.2 INTRODUCTION

In the previous unit, we have learnt about the different types of operating systems. We have also learned about process creation, termination and life cycle of process. We have learnt about creation of threads and multi threading concepts also in the previous units.

In this unit, we will learn about process synchronization and process co-ordination concepts. Race conditions, critical sections problems and their various solutions are explained in detail in this unit. In the next unit, the concept of semaphres are explained in detail.

5.3 PROCESS SYNCHRONIZATION AND CO-ORDINATION

There are many processes that operate concurrently in the operating system. Two or more processes may request the same resource or data. Process synchronization means sharing of system resources by processes in such a way that, concurrent access to shared data is handled thereby minimizing the chance of inconsistent data. To minimize the chance of inconsistent data, certain mechanisms need to be ensured so that data consistency is maintained during execution of processes.

Process coordination is required to maintain synchronization between processes in the operating system. Process synchronization and coordination need to handle problems that arise with multiple process executions. Some of these problems like race condition, critical section and mutual exclusion are discussed in detail in the next few sections.

5.3.1 Race Condition

A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition.**

Examples of Race Condition

As a first example, suppose that two processes, P1 and P2, share the global variable **a**. At some point in its execution, P1 updates **a** to the value 1, and at some point in its execution, P2 updates **a** to the value 2. Thus, the two tasks are in a race to write variable **a**. In this example the “loser” of the race (the process that updates last) determines the final value of **a**.

For our second example, consider two process, P3 and P4, that share global variables **b** and **c**, with initial values b = 1 and c = 2. At some point in its execution, P3 executes the assignment b = b + c,

and at some point in its execution, P4 executes the assignment c = b + c. Note that the two processes update different variables. However, the final values of the two variables depend on the order in which the two processes execute these two assignments. If P3 executes its assignment statement first, then the final values are b = 3 and c = 5. If P4 executes its assignment statement first, then the final values are b = 4 and c = 3.

5.3.2 Critical Sections

Sometimes processes have to access shared memory or files, or doing other critical things that can lead to races. That part of the program where the shared memory is accessed is called the ***critical region*** or ***critical section***.

So, how do we avoid race conditions? We need four conditions to hold to have a good solution to the race condition. They are :

1. No two processes may be simultaneously inside their critical regions.
2. No assumptions may be made about speeds or the number of CPUs.
3. No process running outside its critical region may block other processes.
4. No process should have to wait forever to enter its critical region.

5.4 MUTUAL EXCLUSION

If one process is in the critical region, according to mutual exclusion, no other processes are allowed to enter the critical section at that time. To avoid race condition and to achieve mutual exclusion, let us examine some techniques.

5.4.1 Disabling Interrupts

After entering into the critical region, each process has to disable all interrupts and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur. The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to

another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

A process can then enforce mutual exclusion in the following way

```
while (true) {/* disable interrupts */;
/* critical section */;
/* enable interrupts */;
/* remainder */;}
```

5.4.2 Peterson's Solution

In 1981, G.L. Peterson discovered a much simpler way to achieve mutual exclusion. Peterson's algorithm consists of two procedures written in ANSI C, which means that function prototypes should be supplied for all the functions defined and used.

```
#define FALSE 0
#define TRUE  1
#define N     2     /* number of processes */
int turn;           /* whose turn is it? */
int interested[N]; /*all values initially 0(FALSE*/
void enter_region(int process)/* process is 0 or 1*/
{int other;    /* number of the other process */
other = 1 - process; /* the opposite of process */
interested[process]=TRUE;
 /* show that you are interested */
 turn = process; /* set flag */
 while(turn==process && interested[other]==TRUE);
 /* null statement */}
 void leave_region (int process)
 /* process, who is leaving */
 {interested[process] = FALSE;
 /* indicate departure from critical region */}
```

Before using the shared variables (i.e., before entering its critical region), each process calls *enter_region* with its own process number, 0 or 1, as parameter. This call will cause it to wait, if need be, until it is safe to enter. After it has finished with the shared variables, the process calls *leave_region* to indicate that it is done and to allow the other process to enter, if it so desires.

Let us see how this solution works. Initially neither process is in its critical region. Now process 0 calls *enter_region.* It indicates its interest by setting its array element and sets *turn* to 0. Since process 1 is not interested, *enter_region* returns immediately. If process 1 now calls *enter_region,* it will hang there until *interested*[0] goes to *FALSE*, an event that only happens when process 0 calls *leave_region* to exit the critical region.

Now consider the case that both processes call *enter_region* almost simultaneously. Both will store their process number in *turn.* Whichever store is done last is the one that counts; the first one is overwritten and lost. Suppose that process 1 stores last, so *turn* is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region until process 0 exits its critical region.

5.4.3 Sleep and Wake Up

Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The ***wakeup*** call has one parameter, the process to be awakened. Alternatively, both *sleep* and *wakeup* each have one parameter, a memory address used to match up *sleeps* with *wakeup*s.

The Producer-Consumer Problem

Let us consider one example. Let there are two processes, one is *producer* and the other is *consumer*. Both the processes producer and consumer share a common, fixed-size buffer. The producer, puts information into the buffer, and the other one, the consumer, takes it out.

Trouble arises when the producer wants to put a new item in the buffer, but it is already full. The solution is for the producer to go to sleep, to be awakened when the consumer has removed one or more items. Similarly, if the consumer wants to remove an item from the buffer and sees that the buffer is empty, it goes to sleep until the producer puts something in the buffer and wakes it up.

The code for both producer and consumer is shown below.

```
#define N 100/* number of slots in the buffer*/
int count=0;/* number of items in the buffer*/

void producer (void)
{int item;
while (TRUE) {/* repeat forever */
item = produce_item();/* generate next item */
  if (count == N) sleep();/* if buffer is full,
              go to sleep */
  insert_item(item); /* put item in buffer */
  count = count + 1; /* increment count of
                     items in buffer */
  if (count == 1) wakeup(consumer);
              /* was buffer empty? */}
{void consumer(void)
 int item;
 while (TRUE) { /* repeat forever */
if (count == 0) sleep(); /* if buffer is
                     empty, got to sleep */
  item = remove_item();
              /* take item out of buffer */
  count = count - 1;   /* decrement count of
                     items in buffer */
  if (count == N - 1) wakeup(producer);
                     /* was buffer full? */
  consume_item(item);       /* print item */}
```

To keep track of the number of items in the buffer, we will need a variable, *count*. If the maximum number of items the buffer can hold is *N*, the producer's code will first test to see if *count* is *N*. If it is, the producer will go to sleep; if it is not, the producer will add an item and increment *count*.

The consumer's code is similar: first test *count* to see if it is 0. If it is 0, go to sleep, if it is nonzero, remove an item and decrement the counter. Each of the processes also tests to see if the other should be awakened, and if so, wakes it up.

Now, let us get back to the race condition. It can occur because access to *count* is unconstrained. The following situation could possibly occur. The buffer is empty and the consumer has just read

count to see if it is 0. At that instant, the scheduler decides to stop running the consumer temporarily and start running the producer. The producer inserts an item in the buffer, increments *count*, and notices that it is now 1. Reasoning that *count* was just 0, and thus the consumer must be sleeping, the producer calls *wakeup* to wake the consumer up.

Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of *count* it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever.

5.4.4 Semaphores

In 1965, E. W. Dijkstra (1965) suggested using an integer variable to count the number of wakeups saved for future use. In his proposal, a new variable type, called a ***semaphore***, was introduced. A semaphore could have the value 0, indicating that no wakeups were saved, or some positive value if one or more wakeups were pending.

Dijkstra proposed having two operations, *down* and *up* (generalizations of sleep and wakeup, respectively). The down operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and just continues. If the value is 0, the process is put to sleep without completing the down for the moment. Checking the value, changing it and possibly going to sleep, is all done as a single, indivisible **atomic action**. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. This atomicity is absolutely essential to solving synchronization problems and avoiding race conditions.



CHECK YOUR PROGRESS

Q1. Fill in the blanks:

(a) _____ refers to the communication among the processes without any interruption.

(b) A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called _____.

(c) The part of the shared memory where race condition occurs is called _____

(d) If one process is in the critical region, according to _____, no processes are allowed to enter the critical section at that time.

(e) The producer-consumer problem is a technique to achieve _____.



5.5 LET US SUM UP

- Processes frequently need to communicate with other processes, preferably in a well-structured way not using interrupts. This is termed as ***Interprocess Communication***.
- A situation where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition.**
- Processes have to access shared memory or files, or doing data that can lead to races. The part of the program where the shared memory is accessed is called the ***critical region*** or ***critical section***.
- According to the **mutual exclusion**, no two or more processes are allowed to enter the critical region simultaneously.
- Different types of mutual exclusion techniques are available such as Disabling Interrupt, Sleep and Wake Up, Peterson's solution, Semaphores etc.



5.6 FURTHER READING

1. Tanenbaum, A. S., & Woodhull, A. S. (1987). Operating systems: design and implementation (Vol. 2). Englewood Cliffs, NJ: Prentice-Hall.
2. Silberschatz, A., Galvin, P. B., Gagne, G., & Silberschatz, A. (1998). Operating system concepts (Vol. 4). Reading: Addison-wesley.



5.7 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1. (a) Interprocess Communication

(b) race condition

(c) critical region

(d) mutual exclusion

(e) mutual exclusion



5.8 MODEL QUESTIONS

Q1. What is process co-ordination?

Q2. What is race condition and mutual exclusion?

Q3. What are the different types of mutual exclusion techniques?

Q4. When more than one processes share the same memory location at the same time, what problems may arise?

Q5. What are the requirements to achieve mutual exclusion?

Q6. Explain the disabling interrupt method to achieve mutual exclusion?

Q7. Explain the Peterson's solution to achieve mutual exclusion.

Q8. What is sleep and wakeup? What is semaphore?

Q10. Explain the producer-consumer technique.

****

UNIT 6 : SEMAPHORE

UNIT STRUCTURE

6.1 Learning Objectives
6.2 Introduction
6.3 Semaphores
6.4 Types of Semaphores
6.5 Operations on Semaphore
6.6 Implementation of Semaphore
    6.6.1 Mutual-exclusion Implementation with Semaphores
    6.6.2 Semaphore Implementation with no Busy Waiting
6.7 Application of Semaphores
6.8 Let Us Sum Up
6.9 Further Reading
6.10 Answers to Check Your Progress
6.11 Model Questions

6.1 LEARNING OBJECTIVE

After going through this unit, you would be able to:

- describe semaphore and its types
- learn about the different operations on semaphore
- discuss the applications of semaphore

6.2 INTRODUCTION

In the previous units we have learnt about processes and threads. We have also learnt about the different problems associated with process synchronization and process co-ordination. We have learnt about the concepts of race conditions, mutual exclusion and critical section. In this unit, we will learn about the concept of semaphones. The different types of semaphore, operations and implementations of semaphore are also discussed in detail in this unit. In the next unit, we will explore the working of scheduling algorithms.

6.3 SEMAPHORE

Semaphore, as discussed in Unit 5, is a protected variable or abstract data type used for controling access to a common resource by multiple processes in a concurrent system such as a multiprogramming operating system. In concurrent system if multiple processes share a common resource, they need a way to be able to use that resource without disrupting each other i.e each process has to be able to read from and write to that resource uninterrupted. Thus, the value of semaphore indicates the status of the common resource. It is used to lock the resource that is being used. A process needing the resource checks the semaphore to determine the resource's status and then decides how to proceed. A semaphore will either allow or disallow access to the resource, depending on how it is set up. One example setup would be a semaphore which allowed any number of processes to read from the resource, but only one could ever be in the process of writing to that resource at a time. Semaphore is often used to synchronize operations for multiple processes to avoid starvation and deadlock. Semaphores are commonly used for two purposes: to share a common memory space and to share access to files.

6.4 TYPES OF SEMAPHORE

There are *two types* of semaphore namely *binary semaphore* and *counting semaphore*.

Binary Semaphore: A binary semaphore is one that only takes the values 0 and 1. On some systems, this semaphore is known as **mutex**, which is used by two or more processes to ensure that only one of them can enter its critical region at the same time.

Counting Semaphore: The value of counting semaphore can range over an unrestricted domain(usually 0 to N). Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available.

6.5 OPERATIONS ON SEMAPHORE

Instead of using operations to read and write a shared variable, a semaphore encapsulates the necessary shared data and allows access only by a restricted set of operations. Because a semaphore has the power to suspend and wake processes, semaphores and similar higher-level constructs require the co-operation of the operating system or programming language run-time system. Normally, the semaphore is initially set to the number of available resources. A positive semaphore value indicates that the resource is available. When a process gains the access to a resource, then the value of the semaphore is decremented. The system guarantees that the test and decrement operations on the semaphore are atomic i.e operations are either all performed without interruption or not performed at all. It is guaranteed that once a semaphore operation has started, no other process can access the semaphore until the operation has completed or blocked. For this purpose semaphore is accessed only through two standard operations: ***wait*** and signal.

The wait and signal operations on a semaphore S are as follows-

```
Wait (S) {
        While (S<=0); // do nothing till S<=0
        S—;
         }
```

Wait (S): The wait operation on a semaphore checks to see if the value is greater than 0. If so, it decrements the value (i.e., uses up one stored wakeup) and continues to do so. If the value is 0, the process is put to sleep without completing the down for the moment. The wait operation on semaphore is also called as P operation. The classical definition of wait in pseudo code is given below.

```
Wait (S) {
        While (S<=0); // do nothing till S<=0
        S—;
        }
```

Signal(S): The signal operation increments the value of the semaphore addressed. If one or more processes are sleeping on that semaphore and are unable to complete an earlier wait operation, one of them is chosen by the system (e.g., at random) and is allowed to complete its wait operation. The signal operation on semaphore is also called V operation. The classical definitions of signal in pseudo code is given below.

```
Signal (S) {
        S++;
        }
```

6.6 IMPLEMENTATION OF SEMAPHORE

6.6.1 Mutual-exclusion implementation with semaphores

Let there be n-processes and let them share a binary semaphore (mutex) initialized to 1. Each process P is organized as shown below:

```
do      {
        wait (mutex);
        critical_section
        signal (mutex);
        non_critical_section.
        } while (1);
```

Disadvantage: Mutual-exclusion solutions given by semaphores require busy waiting. That is, while a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. Hence, busy waiting wastes CPU cycles that some other process might be able to use productively.

Advantage: This type of semaphore is also called *spinlock* because the process “spins” while waiting for the lock. Spinlocks are useful in multiprocessor systems as no context switch is required when a process must wait on a lock. Thus, when locks are expected to be held for short times, spinlocks are useful.

There are some interprocess communication primitives that block instead of wasting CPU time when they are not allowed to enter their critical regions. One of the simplest is the pair *sleep* and *wakeup.* Sleep is a system call that causes the caller to block, that is, be suspended until another process wakes it up. The wakeup call has one parameter, the process to be awakened. Alternatively, both sleep and wakeup each has one parameter, a memory address used to match up sleeps with wakeups.

6.6.2 Semaphore Implementation with no busy waiting

- With each semaphore there is an associated waiting queue.
- Each entry in a waiting queue has two data items:
  - value (of type integer)
  - pointer to next record in the list
- Two operations:
  - sleep: place the process invoking the operation on the appropriate waiting queue
  - wakeup : remove one of processes from the waiting queue and place it in the ready queue

```
typedef struct
        {
           int value;
           struct process *list;
        } semaphore;
Wait (semaphore *S)
     {
          S->value—;
          if (S->value < 0)
          {
        add the current process to S->list;
           sleep();
           }
     }
```

```
Signal (semaphore *S)
       {
              S->value++;
              if (S->value <= 0)
               {
                 remove a process P from S->list;
                 wakeup (P);
               }
             }
```

6.7 APPLICATIONS OF SEMAPHORES

One of the applications of semaphore is to solve producer consumer problem. We already know about the producer– consumer problem from unit 5.

The Producer-Consumer Problem:

In producer-consumer problem two processes share a common, fixed N size buffer, where N denotes the maximum number of information that the buffer can hold. One of them, the producer, which puts information into the buffer, and the other one is the consumer, takes the information out. (The generalize version of this problem has m producers and n consumers). We can use a variable 'count' to keep track of the number of items in the buffer. If the producer wants to put a new item in the buffer then it will first test to see if count is N. If it is, the producer will go to sleep. It will be awakened when the consumer has removed one or more items. If count is less than N then the producer will add an item and increment count. The consumer code is similar. If consumer wants to remove an item from the buffer then it will first test count to see if it is 0. If it is, go to sleep until the producer puts something in the buffer and wakes it up. If count is nonzero, it will remove an item and decrement the value of count.

Now, the following situation could possibly occur (race condition). The buffer is empty and the consumer has just read count to see if it is 0. At that time, the scheduler decides to stop running the consumer temporarily and start

running the producer. The producer inserts an item in the buffer, increments count, and notices that it is now 1. Reasoning that count was just 0, and thus the consumer must be sleeping, the producer calls wakeup to wake the consumer up.

Unfortunately, the consumer is not yet logically asleep, so the wakeup signal is lost. When the consumer next runs, it will test the value of count it previously read, find it to be 0, and go to sleep. Sooner or later the producer will fill up the buffer and also go to sleep. Both will sleep forever. The algorithm for producer consumer problem with race condition is given in algorithm 6.1.

Algorithm 6.1: The producer-consumer problem with a fatal race condition

```
 #define N 100        /*number of slots in the buffer*/
 int count = 0; /*number of items in the buffer*/
 void producer(void)
{
int item;
while (TRUE) /*repeat forever*/
      {
      item = produce item( );     /*generate next item*/
      if (count == N) sleep( );
                      /*if buffer is full, go to sleep*/
      insert item(item);  /*put item in buffer*/
      count = count + 1;
                /*increment count of items in buffer*/
      if  (count==1)  wakeup(consumer);/*was  buffer
empty?*/
      }
}
void consumer(void)
{
int item;
while (TRUE)      /*repeat forever*/
      {
       if (count == 0) sleep( );
                /*if buffer is empty, got to sleep*/
```

```
      item = remove item( ); /*take item out of buffer*/
      count = count”1;
                /*decrement count of items in
buffer*/
      if (count == N”1) wakeup(producer);
                /*was buffer full?*/
       consume item(item);    /*print item*/
}
```

Solving the Producer Consumer Problem using Semaphores:

This solution uses three semaphores:

First one is called *full* for counting the number of slots that are full. Initially it is 0. *Second* one is called *empty* for counting the number of slots that are empty. It is initially equal to the number of slots in the buffer. *Third* one is called *mutex* to make sure the producer and consumer do not access the buffer at the same time. Initially it is set at 1.

If each process does a wait operation just before entering its critical region and a signal operation just after leaving it, mutual exclusion is guaranteed. The mutex semaphore is used for mutual exclusion. It is designed to guarantee that only one process at a time will be reading or writing the buffer and the associated variables. The other use of semaphores is for synchronization. The full and empty semaphores are needed to guarantee that certain event sequences do or do not occur. In this case, they ensure that the producer stops running when the buffer is full, and that the consumer stops running when it is empty. This use is different from mutual exclusion. The algorithm for producer consumer problem using semaphores is given in algorithm 6.2.

Algorithm 6.2: The producer-consumer problem using semaphores

```
#define N 100     /*number of slots in the buffer*/
typedef int semaphore; /*semaphores are a special kind
of int*/
semaphore mutex = 1;/*controls access to critical region*/
semaphore empty = N; /*counts empty buffer slots*/
semaphore full = 0;  /*counts full buffer slots*/
void producer(void)
```

```
{
int item;
while (TRUE)                /*TRUE is the constant 1*/
      {
      item = produce item( );
                  /*generate something to put in buffer*/
      wait(&empty);         /*decrement empty count*/
      wait(&mutex);         /*enter critical region*/
      insert item(item);    /*put new item in buffer*/
      signal(&mutex);       /*leave critical region*/
      signal(&full);  /*increment count of full slots*/
      }
}
void consumer(void)
{
int item;
while (TRUE)              /*infinite loop*/
      {
      wait(&full);   /*decrement full count*/
      wait(&mutex); /*enter critical region*/
      item = remove item( );  /*take item from buffer*/
      signal(&mutex); /*leave critical region*/
      signal(&empty); /*increment count of empty slots*/
      consume item(item); /*do something with the item*/
      }
}
```



CHECK YOUR PROGRESS

Write true or false for the following-

Q1. Semaphore is an integer variable.

Q2. The wait operation is also called as V and signal operation is also called as P.

Q3. Semaphore which is used to guarantee mutual exclusion is called mutex.

Q4. Mutex is a counting semaphore.

Q5. Producer consumer problem has one fixed sized buffers.

Q6. In producer consumer problem producer puts item in to a buffer and consumer remove items from the buffer.



6.8 LET US SUM UP

- Semaphore always has a non-negative integer value (i.e., 0, 1, ...).
- It can be initialized to a non-negative integer value.
- A process can do a P operation (or “wait”) on S, denoted P(S). P(S) atomically decrements S by 1 only when S>0 holds; the process waits until S>0 holds.
- A process can do a V operation (or “signal”) on S, denoted V(S). V(S) atomically increments S by 1.
- Semaphore is two of types– Binary semaphore and Counting semaphore
- Value of binary semaphore is either 0 or 1. Values of counting semaphore can be 0 up to N.
- We can use semaphore to solve producer consumer problem.



6.9 FURTHER READING

1. Silberschatz, A., Galvin, P. B., Gagne, G., & Silberschatz, A. (1998). *Operating system concepts* (Vol. 4). Reading: Addison-wesley.

2. Dhamdhere, D. M. (2006). *Operating Systems: A Concept-based Approach, 2E*. Tata McGraw-Hill Education.

3. Tanenbaum, A. S., & Woodhull, A. S. (1987). *Operating systems: design and implementation* (Vol. 2). Englewood Cliffs, NJ: Prentice-Hall.



6.10 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: True
Answer to Q2: False
Answer to Q3: True
Answer to Q4: False
Answer to Q5: True
Answer to Q6: True



6.11 MODEL QUESTIONS

Q1. What is semaphore?
Q2. What are the different types of semaphore available?
Q3. What are the operations that are used to access a semaphore?
Q4. Give the code to implement a semaphore.
Q5. What is producer consumer problem? How can we solve it using semaphore? Can we solve it without using semaphore? Justify your answer.
Q6. What is sleep and wakeup system calls?

UNIT 7 : SCHEDULING

UNIT STRUCTURE

7.1 Learning Objectives
7.2 Introduction
7.3 Scheduling Objectives
7.4 Process Scheduling Queues
7.5 Nonprimitive and Primitive Scheduling
7.6 Types of Schedulers
7.7 Scheduling Performance Criteria
7.8 Scheduling Algorithms
7.9 Let Us Sum Up
7.10 Further Reading
7.11 Answers to Check Your Progress
7.12 Model Questions

7.1 LEARNING OBJECTIVES

After going through this unit, you will be able to :

- learn about the basic concept of scheduling
- differentiate between primitive and nonprimitive scheduling
- identify the types of scheduling
- discuss different scheduling algorithms

7.2 INTRODUCTION

In the previous units, we have learnt about the different concepts of process and threads. We have also learnt about the different problems associated with process synchronization and co-ordination. We have in the previous unit, explored the concept of semaphores in detail. The different types and operations on semaphore had been discussed in detail. In this unit, we will learn about the concept of scheduling. Different types of scheduling algorithms are explained in detail in this unit. In the next unit, we will explore the concept of deadlock.

7.3 SCHEDULING OBJECTIVES

Scheduling is a basic operating system function. All computer resources are scheduled before use. Since CPU is one of the primary computer resources, its scheduling is central to operating system design. **Scheduling** refers to a set of policies and mechanisms supported by operating system that controls the order in which the work to be done is completed. A **scheduler** is an operating system program that selects the next job to be admitted for execution.

A scheduling discipline should try to satisfy the following objectives :–

- **maximize throughput** – A scheduling discipline should attempt to service the largest possible number of processes per unit time.
- **be fair** – A scheduling algorithm is fair if all processes are treated the same, and no process should wait indefinite time from being executed.
- **balance resource use** – The scheduling mechanisms should keep the resources of the system busy.
- **enforce priorities** – In a priority based processes, the scheduling mechanism should favour the higher priority processes.

7.4 PROCESS SCHEDULING QUEUES

A system may have several processes. These processes are stored into a linear list called **job queue**. This queue consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue**. This queue is generally stored as a link list. The operating system also maintains some other queues. For example, when a process is allocated the processor for execution and after executing a moment it interrupts for other resources such as device(i.e. disk) request then the process is kept in a queue called **device queue**. In other words, a device queue is a list of suspended processes that are waiting for I/O devices such as a disk. A new process initially placed in the ready queue for execution. It waits in the ready queue until it is selected for execution.

7.5 NONPRIMITIVE and PRIMITIVE SCHEDULING

All scheduling algorithms are broadly categorised into two groups i.e. (a) nonprimitive and (b) primitive.

(a) **Nonprimitive Scheduling** : A scheduling algorithm is said to be nonpreemptive if, once a process has been given to the CPU, the CPU cannot be taken away from that process at any condition**.**

In nonpreemptive scheduling, a schedular executes jobs in the following two situations.

(i) When a process switches from running state to the waiting state.

(ii) When a process terminates.

(b) **Preemptive Scheduling** : A scheduling algorithm is said to be preemptive if, once a process has been given the CPU, it can be taken away during the execution for other processes.

7.6 TYPES OF SCHEDULERS

As discussed earlier, a scheduler is an operating system program to be selected the next process for execution. During the lifetime of a process it switches between the various scheduling queues. For example, a process may migrate from job queue to ready queue, ready queue to device queue, device queue to ready queue and so on. The process of migration is carried out by the appropriate scheduler. Based on this working principle, schedulers are classified as follows:

(i) Long term scheduler or job scheduler.

(ii) Medium term scheduler.

(iii) Short term scheduler.

(i) Long term scheduler :

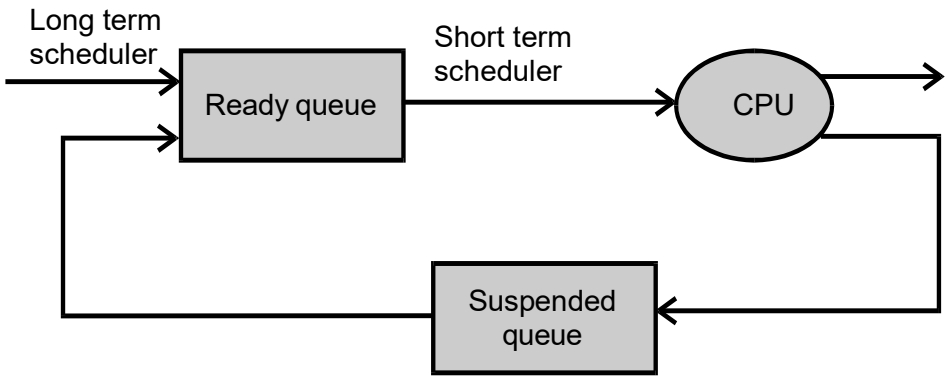


Fig. 7.1 : Long term and short term schedulers

The objective of multiprogramming is to have some process running at all times, so as to maximize CPU utilization. There are always more processes than it can be executed by CPU. These processes are stored in secondary memory for later processing. The **long term scheduler** selects processes from secondary memory and load them into main memory. The **short-term scheduler** or **CPU scheduler** selects a process from the ready queue and allocates the CPU for execution. The long-term scheduler executes less frequently than the short-term scheduler. If the average rate of number of processes arriving in memory is equal to that of departuring the system then the long term scheduler may need to be invoked only when a process departs the system. Because of longer time taken by CPU during execution, the long-term scheduler can spend more time to select a process for execution. The long-term scheduler should take a careful selection of processes. The processes should be combination of CPU and I/O bounds type. If all processes are I/O bound, the ready queue will always be empty. If all processes are CPU bound, no process will be waiting for I/O operation and the system will be unbalanced. Therefore, the system performance depends on the selection of processes by the long-term scheduler.

(ii) Medium term scheduler :

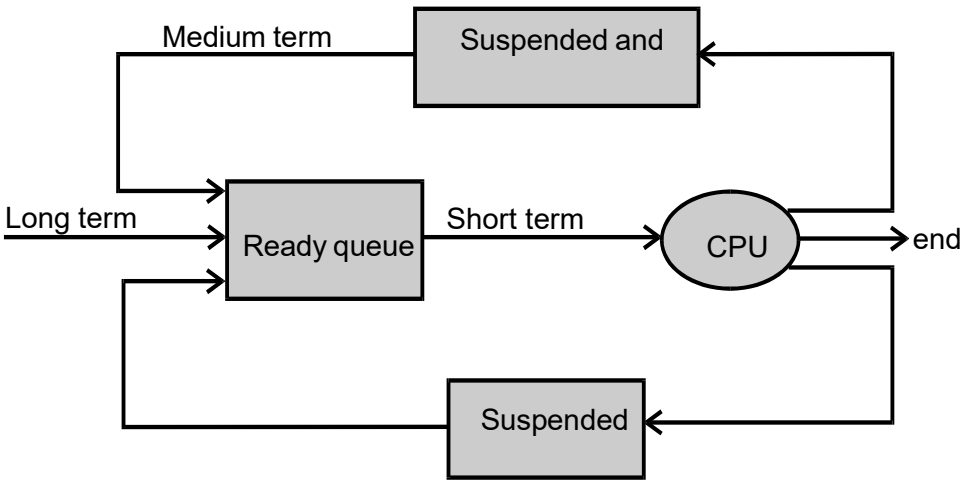


Fig.7.2 : Long term, medium term and short term schedulers

A process may be suspended during its execution due to some I/O operation. In most cases, processes are suspended for I/O operation, during its operation. It is beneficial to remove these processes from main memory to secondary storage device to make room for other processes. At some later time these processes can be reloaded into the ready queue and continued where from it were left previously. The process of transferring a process from main memory to secondary memory is called swapped-out and the reverse is called swapped-in. Swapped-in and swapped-out is done by **medium-term scheduler**.

(iii) Short-term scheduler

A short-term scheduler allocates processes from ready queue to CPU. Its main objective is to maximize CPU requirement. Compared to the other two schedulers it is more frequent. It must select a new process for execution quite often because a CPU executes a process only for a few millisecond before it goes for I/O operation.

7.7 SCHEDULING PERFORMANCE CRITERIA

The performance criteria that are frequently used by schedulers to maximize the system performance are

(i) CPU Utilisation
(ii) Throughput
(iii) Waiting time
(iv) Turnaround time
(v) Response time.

(i) CPU Utilisation : The idea is that, the CPU should be busy as far as possible. CPU utilisation may range from 0 to 100 percent.

(ii) Throughput : Throughput means number of processes completed per time unit. The higher the number of processes, the more work being done by the system.

(iii) Waiting time : The objective of multiprogramming operating system is to keep CPU busy for all the time. As such several processes are waiting in the ready queue of memory for execution. The CPU can

execute only one job at a time. The rest of the job waits for the CPU. A process may be in the ready queue several times during the lifetime of a process. It depends on the number of interrupts for other resources such as I/O request. The waiting time of a process may be expressed as the sum of the time waiting in the ready queue during the lifetime of a process. Waiting time does not count the CPU time of a process. It can be expressed as

Waiting time=Turnaround time – processing time

(iv) Turnaround time : The time period from the submission of a process to the time of its completion is called turnaround time. It is the sum of the time spent waiting to get into memory, waiting in the ready queue, CPU time and I/O time.

(v) Response time : A process may produce some intermediate result to the user before the execution is completed. In such a situation, the measure of time from the submission of a request until the first response is produced is called response time.

The main objectives of designing scheduler are to maximize CPU utilization and throughput, and to minimize turnaround time, waiting time and response time. One of the problems in designing scheduler and selecting a set of its performance criterias is that they often conflict with each other. For example, throughput and CPU utilisation may be increased by executing the large number of processes, but then response time may suffer. Therefore, the design of a scheduler usually requires balance of all the different requirements and constraints.

7.8 SCHEDULING ALGORITHM

Scheduling algorithms are concerned with the selection of a process from the ready queue to be allocated to the CPU. There are several scheduling algorithms with different working principle. All scheduling algorithms can be divided into two categories-

(i) pre-emptive and

(ii) non-preemptive scheduling.

Non-preemptive scheduling means, if once a process has been given the CPU, the CPU can not be taken away from that process. A scheduling is

pre-emptive if the CPU can be taken away. There are four conditions where a pre-emptive scheduling can take decisions-

(i) When a process migrates from the running state to the waiting state.

(ii) When a process migrates from the waiting state to the ready state.

(iii) When a process migrates from the running state to the ready state.

(iv) When a process terminates.

When scheduling takes place only under condition (i) & (iv), we say the scheduling scheme is nonpreemptive, otherwise the scheduling scheme is preemptive.

Preemptive scheduling is more useful in high priority process which requires immediate response, but it is more costly than nonpreemptive scheduling. The following are some common CPU-scheduling algorithms-

(i) First-Come, First –Served Scheduling

(ii) Shortest-Job-First Scheduling

(iii) Priority Scheduling

(iv) Round-Robin Scheduling

(v) Multilevel Queue Scheduling

(vi) Multilevel Feedback Queue Scheduling

(i) First-Come, First-Served(FCFS) Scheduling : This scheduling algorithm can be compared with the people waiting in front of a railway reservation counter. The reservation clerk serves the first person waiting on the queue, then the second one and so on. In FCFS, processes are stored into a ready queue, then allocated into CPU on first come fist serve basis for execution.

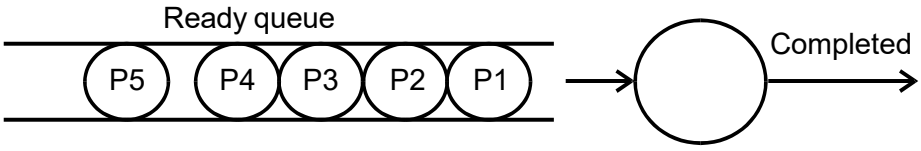


Fig.7.3 : First-Come-First-served Scheduling

Advantages of FCFS :

- The code for FCFS scheduling is simple to write and understand.
- The implementation of FCFS policy is easily managed with a FIFO queue.

Disadvantages of FCFS :

- A first-come-first-served scheduling is nonpreemptive which usually results in poor CPU performance.
- Short jobs may suffer considerable turnaround delays and waiting times when CPU has been allocated to longer jobs.
- FCFS algorithm is particularly troublesome for time-sharing systems, where each users needs to get a share of the CPU at regular intervals.
- The average waiting time is large.

Example 7.1

Consider the following three processes that arrive at time zero(0), with the CPU-burst time given in milliseconds:

Process	Burst Time(Execution time or CPU time)
P1	20
P2	4
P3	3

If the processes arrive in the order P1, P2, and P3 and are served in FCFS order, then the Gantt chart will be:

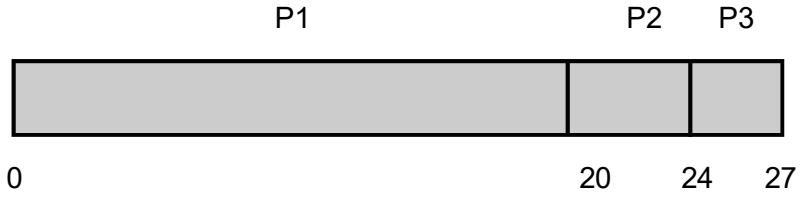


Fig.7.4 : FCFS Gantt chart

The waiting time is 0 millisecond for process P1, 20 milliseconds for process P2, and 24 milliseconds for process P3. Thus, the average waiting time is (0+20+24)/3 =14.66 milliseconds.

(ii) Shortest-Job-First (SJF) Scheduling : This algorithm depends on the execution time(or burst time) of each processes. When a CPU is available, it is assigned to the process that has the smallest next CPU execution time. If two processes have the same execution time then FCFS is followed.

Advantages of SJF :

- It gives the minimum average waiting time for a given set of processes.

Disadvantages of SJF :

- It requires the knowledge of execution time of each process in advance.
- It may not be implemented at the level of short-term scheduling, as there is no way to know the length of the next CPU burst.

Example 7.2

Consider the processes shown in example 7.1. The Gantt chart for these processes is shown as bellow.

Process	Burst Time(Execution time or CPU time)
P1	20
P2	4
P3	3

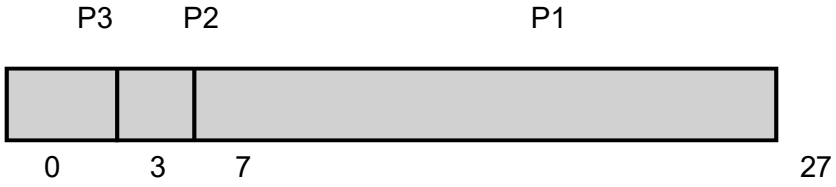


Fig.7.5 : SJF Gantt chart

The waiting time is 7 millisecond for process P1, 3 milliseconds for process P2, and 0 milliseconds for process P3. Thus, the average waiting time is (7+3+0)/3 =3.33 milliseconds. If we were using the FCFS scheduling scheme, then the average waiting time would be 14.66 milliseconds. Hence, SJF reduces the considerable amount of average waiting time over FCFS.

(iii) Priority Scheduling : This algorithm depends on the priority of each process. A priority number is associated with each process, and the CPU is allocated to the process with the highest priority. There is no specification whether 0 is the highest or lowest priority. Some systems use 0 as low priority, others use 0 as high priority. Equal priority processes are scheduled in FCFS order. Here, we will use low numbers to represent high priority. Priorities can be defined either internally or externally. Internal priorities are defined based on some internal factors of a process such as execution time, memory requirements, the number of open files. External priorities are defined based on some external factors such as importance of the process,

amount of fund for the process etc. There may be two cases for allocating a process to the CPU.

(a) When a CPU is available, it is allocated to the process that is in the ready queue that has the highest next priority.

(b) When a process arrives at the ready queue, its priority is compared with the priority of the currently running process. If the priority of the newly arrived process is higher than the priority of the currently running process then a preemptive priority scheduling algorithm will preempt the CPU. A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.

Advantages

- Priority scheduling can be either preemptive or nonpreemptive.
- Higher priority processes get preference for execution, even if it is entered last in the ready queue.

Disadvantages

- Average waiting time may be high.
- It can leaves some low priority process waiting indefinitely(starvation) for the CPU.

Example 7.3

Consider the processes shown below. The Gantt chart for these processes is shown as bellow.

Process	CPU time	Priority
P1	20	3
P2	4	1
P3	3	2

P2 P3 P1



Fig.7.6 : Priority scheduling Gantt chart

Here, we assumed that the process have arrived at time 0, in the order P1, P2 and P3, with the length of the CPU time in milliseconds. The average waiting time is (7+0+4)/3=3.66 milliseconds.

(iv) Round-Robin Scheduling : It is basically used in a time-sharing and a multi-user system. The basic principle is that the CPU time is divided into time slices. Each process is allocated a small time slice(10-100 milliseconds) while it is running. No process can run for more than one time slice when there are others waiting in the ready queue. If a process needs more CPU time to complete, it goes to the end of ready queue to await the next allocation. If a running process releases the CPU voluntarily due to I/O request or completion of execution, another process is scheduled to run.

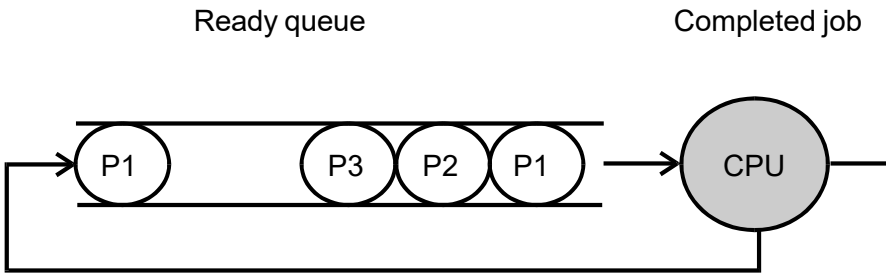


Fig.7.7 : Round Robin Scheduling

Advantages.

- It utilises the system resources in an equitable manner.
- Small process may be executed in a single time slice giving good response time.

Disadvantages.

- Average waiting time may be high.
- Turnaround time depends on the size of the time quantum.

Example 4

Consider the processes shown in example 7.1.

Process	CPU time
P1	20
P2	4
P3	3

One can use a time slice of 4 milliseconds. In such case, process P1 gets the first 4 milliseconds. Since it requires another 16 milliseconds, it is pre-empted after the first time slice, and the CPU is allocated to the next

process in the ready queue, process P2. Since process P2 needs exactly 4 milliseconds, it quits from the system. The CPU is then allocated to the next process, process P3. Since process P3 does not need 4 milliseconds, it quits before its time slice expires. Once each process has received one time slice, the CPU is allocated to process P1 for an additional time slice. The Gantt chart for these processes is shown as bellow.

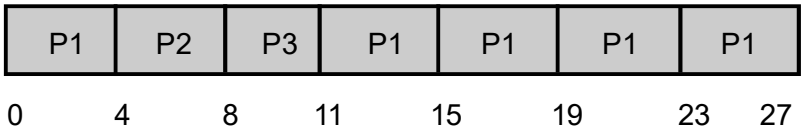


Fig.7.8 : Round Robin scheduling Gantt chart

The average waiting time is (0+11+4+8)/3=7.66 milliseconds.

(v) Multilevel Queue Scheduling : A multilevel queue scheduling algorithm partitions the ready queue into separate queue. Processes are permanently assigned to each queue, usually based upon properties such as memory size, process priority or process type. Each queue has its own scheduling algorithm. The interactive queue might be scheduling by a round-robin algorithm while batch queue follows FCFS. As an example of multiple queues scheduling one simple approach to partitioning of the ready queue into system processes, interactive processes, interactive editing processes, batch processes and student processes which create a five ready queue Fig.7.9. Each queue is serviced by some scheduling algorithm best suited to the type of processes stored in the queue.

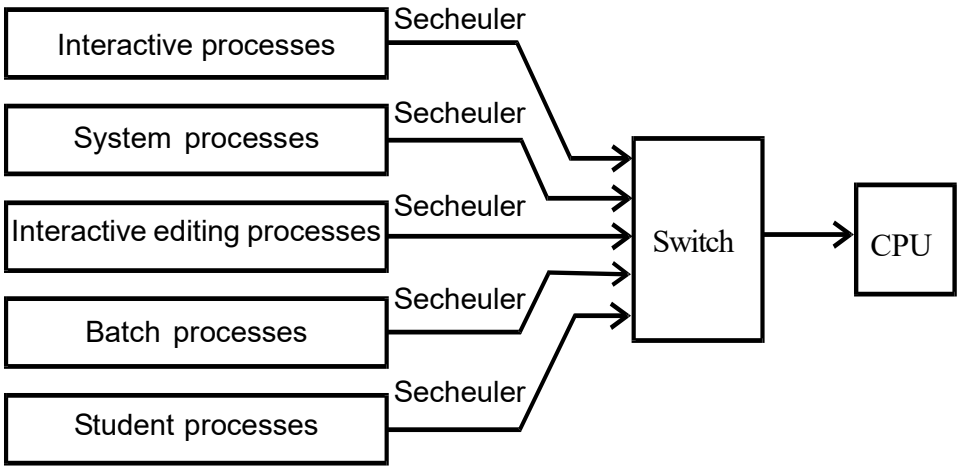


Fig.7.9 : Multilevel queue scheduling



CHECK YOUR PROGRESS

Q1. The processes that stored into a liner list in a computer system are called _____.

Q2. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called _____.

Q3. A device queue is a list of _____ that are waiting for I/O devices such as a disk.

Q4. What is a scheduler?

Q5. Define long term and short term scheduler.



7.9 LET US SUM UP

- Scheduling is the process of allocating processes to the CPU.
- A good scheduling technique should - have **maximize throughput, be fair, balance resource use, enforce priorities**.
- The following are some common CPU-scheduling algorithms-
  - First-Come, First –Served Scheduling
  - Shortest-job-First Scheduling
  - Priority Scheduling
  - Round-Robin Scheduling
  - Multilevel Queue Scheduling
  - Multilevel Feedback Queue Schedulin



7.10 FURTHER READING

1. Silberschatz, A., Galvin, P. B., Gagne, G., & Silberschatz, A. (1998). Operating system concepts (Vol. 4). Reading: Addison-wesley.
2. Tanenbaum, A. S., & Woodhull, A. S. (1987). Operating systems: design and implementation (Vol. 2). Englewood Cliffs, NJ: Prentice-Hall.



7.11 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1. job queue

Answer to Q2. ready queue

Answer to Q3. suspended processes

Answer to Q4. A scheduler is an operating system program to be selected the next process for execution.

Answer to Q5. The **long term scheduler** selects processes from secondary memory and load them into main memory. The **short-term scheduler** or **CPU scheduler** selects a process from the ready queue and allocates the CPU for execution.



7.12 MODEL QUESTIONS

Q1. Explain primitive and nonprimitive scheduling.

Q2. A CPU-scheduling algorithm determines an order for the execution of its scheduled processes. Given n processes to be scheduled on one processor, how many different schedules are possible? Explain.

Q3. Explain short-term, medium-term and long-term schedulers. Which one controls the degree of multiprogramming?

Q4. Consider the following set of processes, with the length of CPU-burst time given in milliseconds:

Process	Burst time	Priority
P1	16	3
P2	1	1
P3	6	2
P4	2	3
P5	5	4

The processes are assumed to have arrived in the order P1, P2, P3, P4, P5, all at time 0.

(i) Draw Gantt charts illustrating the execution of these processes using FCFS, SJF, Priority and Round-Robin(quantum=4) scheduling.

(ii) What is the turnaround time of each process for each of the scheduling algorithms in (i).

(iii) What is the waiting time of each process for each of the scheduling algorithms in (i).

(iv) Which of the schedules in part (i) results in the minimal average waiting time(over all processes).

UNIT 8 : DEADLOCKS

UNIT STRUCTURE

8.1 Learning Objectives
8.2 Introduction
8.3 Deadlock
8.4 Principles of Deadlock
    8.4.1 Deadlock Conditions
    8.4.2 Deadlock Modeling
8.5 Deadlock Prevention
8.6 Deadlock Detection and Recovery
    8.6.1 Deadlock Detection
    8.6.2 Deadlock Recovery
8.7 Deadlock Avoidance
    8.7.1 Banker's Algorithm
8.8 Let Us Sum Up
8.9 Further Reading
8.10 Answers To Check Your Progress
8.11 Model Questions

8.1 LEARNING OBJECTIVES

After going through this unit, you will be able to:

- describe deadlock
- learn about different methods for preventing or avoiding deadlocks in a computer system.

8.2 INTRODUCTION

In the previous units we have learnt about semaphores and scheduling algorithms. We have discussed the different types of semaphore and implementations of semaphore. We have also studied various scheduling algorithms in the previous unit like first come first serve, round robin, shortest job first and priority scheduling. In this unit the concepts of deadlock is explored.

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change the state, because the resources it has requested are held by other waiting processes. This situation is called a *deadlock*. This unit describes the methods that an operating system can use to prevent or deal with deadlocks. This is the last unit of this block. In the next block, we will explore concepts like memory management and I/O management.

8.3 DEADLOCK

Deadlock can be defined as the *permanent* blocking of a set of processes that either compete for system resources or communicate with each other. A set of processes are deadlocked when each process in the set is blocked waiting for an event (typically the freeing up of some requested resource) that can only be triggered by another blocked process in the set. Deadlock is permanent because none of the events is ever triggered. Unlike other problems in concurrent process management, there is no efficient solution in the general case. In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

8.4 PRINCIPLES OF DEADLOCK

Two general categories of resources can be distinguished: *reusable* and *consumable*. A reusable resource is one that can be safely used by only one process at a time and is not depleted by that use. Processes obtain resource units that they later release for reuse by other processes. Examples of reusable resources include processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases etc.

A consumable resource is one that can be created (produced) and destroyed (consumed). Typically, there is no limit on the number of consumable resources of a particular type. An unblocked producing process may create any number of such resources. When a resource is acquired by a consuming process, the resource ceases to exist. Examples of consumable resources are interrupts, signals, messages, and information in I/O buffers.

8.4.1 Deadlock Conditions

Now, let us now discuss the conditions that must be present for a deadlock to occur. These conditions are:

1. **Mutual exclusion condition :** Each resource is either currently assigned to exactly one process or is available.
2. **Hold and wait condition :** Processes that are currently holding resources that were granted earlier can request new resources.
3. **No preemption condition :** Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. **Circular wait condition :** There must be a circular chain of two or more processes, each of which is waiting for a resource held by the next member of the chain.

8.4.2 Deadlock Modeling

In 1972, Holt showed how directed graphs can be used to model these four conditions. The graphs have two kind of nodes: processes, shown as circles, and resources, shown as squares as in Fig. 8.1. Here it is assumed that there is one resource of each kind.

An arc from a resource to a process means that the resource has previously been requested by, granted to, and is currently held by that process. In Fig. 8.1(b), resource *Ra* is currently assigned to process P1.

An arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. 8.1 (a), process P2 is waiting for resource Ra.

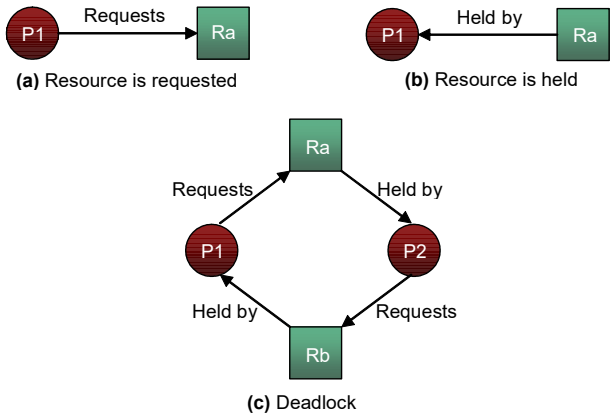


Fig. 8.1: Resource allocation

Fig. 8.1 (c) represents a deadlock: process P1 is waiting for resource Ra, which is currently held by process P2. Process P2 is not about to release resource Ra because it is waiting for resource Rb, held by P1. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle. In this example, the cycle is P1–Ra–P2–Rb–P1.

8.5 DEADLOCK PREVENTION

The strategy of deadlock prevention is, simply put, to design a system in such a way that the possibility of deadlock is excluded. In general, the deadlock prevention methods fall into two classes. An indirect method of deadlock prevention is to prevent the occurrence of one of the three necessary conditions listed previously (condition 1 through 3). A direct method of deadlock prevention is to prevent the occurrence of a circular wait (condition 4). Now let's examine techniques related to each of the four conditions.

- **Mutual Exclusion:**

In general, the first of the four listed conditions cannot be disallowed. If access to a resource requires mutual exclusion, then mutual exclusion must be supported by the operating system. Some resources, such as files, may allow multiple accesses for reads but only exclusive access for writes. Even in this case, deadlock can occur if more than one process requires write permission.

- **Hold and Wait:**

The hold-and-wait condition can be prevented by requiring that a process request all of its required resources at one time and blocking the process until all requests can be granted simultaneously. This approach is inefficient in two ways. First, a process may be held up for a long time waiting for all of its resource requests to be filled, when in fact it could have proceeded with only some of the resources. Second, resources allocated to a process may remain unused for a considerable period, during which time they are denied to other processes. Another problem is that a process may not know in advance all of the resources that it will require.

There is also the practical problem created by the use of modular programming or a multithreaded structure for an application. An application would need to be aware of all resources that will be requested at all levels or in all modules to make the simultaneous request.

- **Nopreemption:**

This condition can be prevented in several ways. First, if a process holding certain resources is denied a further request, that process must release its original resources and, if necessary, request them again together with the additional resource. Alternatively, if a process requests a resource that is currently held by another process, the OS may preempt the second process and require it to release its resources. This latter scheme would prevent deadlock only if no two processes possessed the same priority.

This approach is practical only when applied to resources whose state can be easily saved and restored later, as is the case with a processor.

- **Circular Wait :**

The circular-wait condition can be prevented by defining a linear ordering of resource types. If a process has been allocated resources of type *R*, then it may subsequently request only those resources of types following *R* in the ordering.

To see that this strategy works, let us associate an index with each resource type. Then resource *Ri* precedes *Rj* in the ordering if $i < j$. Now suppose that two processes, A and B, are deadlocked because A has acquired *Ri* and requested *Rj*, and B has acquired *Rj* and requested *Ri*. This condition is impossible because it implies $i < j$ and $j < i$.

As with hold-and-wait prevention, circular-wait prevention may be inefficient, slowing down processes and denying resource access unnecessarily.

8.6 DEADLOCK DETECTION AND RECOVERY

8.6.1 Deadlock Detection

Deadlock prevention strategies are very conservative; they solve the problem of deadlock by limiting access to resources and by imposing restrictions on processes. At the opposite extreme, deadlock detection strategies do not limit resource access or restrict process actions. With deadlock detection, requested resources are granted to processes whenever possible. Periodically, the operating system performs an algorithm that allows it to detect the circular wait condition described earlier in condition 4 and illustrated in Fig. 8.1.

A check for deadlock can be made as frequently as each resource request or, less frequently, depending on how likely it is for a deadlock to occur. Checking at each resource request has two advantages: it leads to early detection, and the algorithm is relatively simple because it is based on incremental changes to the state of the system. On the other hand, such frequent checks consume considerable processor time.

8.6.2 Deadlock Recovery

Once deadlock has been detected, some strategy is needed for recovery. The following are possible approaches, listed in order of increasing sophistication:

1. Abort all deadlocked processes. This is, believe it or not, one of the most common, if not the most common, solution adopted in operating systems.
2. Back up each deadlocked process to some previously defined checkpoint, and restart all processes. This requires that rollback and restart mechanisms be built in to the system. The risk in this approach is that the original deadlock may recur. However, the nondeterminancy of concurrent processing may ensure that this does not happen.
3. Successively abort deadlocked processes until deadlock no longer exists. The order in which processes are selected for abortion should be on the basis of some criterion of minimum cost. After each abortion, the detection algorithm must be reinvoked to see whether deadlock still exists.
4. Successively preempt resources until deadlock no longer exists. As in (3), a cost based selection should be used, and reinvocation of the detection algorithm is required after each preemption. A process that has a resource preempted from it must be rolled back to a point prior to its acquisition of that resource.

For (3) and (4), the selection criteria could be one of the following. Choose the process with the –

- least amount of processor time consumed so far
- least amount of output produced so far
- most estimated time remaining
- least total resources allocated so far
- lowest priority

Some of these quantities are easier to measure than others. Estimated time remaining is particularly difficult. Also, other than by means of the priority measure, there is no indication of the "cost" to the user, as opposed to the cost to the system as a whole.

8.7 DEADLOCK AVOIDANCE

An approach to solving the deadlock problem that differs subtly from deadlock prevention is deadlock avoidance. In deadlock prevention, we constrain resource requests to prevent at least one of the four conditions of deadlock. This is either done indirectly, by preventing one of the three necessary policy conditions (mutual exclusion, hold and wait, no preemption), or directly by preventing circular wait. This leads to inefficient use of resources and inefficient execution of processes. Deadlock avoidance, on the other hand, allows the three necessary conditions but makes judicious choices to assure that the deadlock point is never reached. As such, avoidance allows more concurrency than prevention. With deadlock avoidance, a decision is made dynamically whether the current resource allocation request will, if granted, potentially lead to a deadlock. Deadlock avoidance thus requires knowledge of future process resource requests.

In this section, we describe two approaches to deadlock avoidance:

- Do not start a process if its demands might lead to deadlock.
- Do not grant an incremental resource request to a process if this allocation might lead to deadlock.

8.7.1 Banker's Algorithm

In 1965, Dijkstra proposed a scheduling algorithm that can avoid deadlocks, known as the *Banker's algorithm*. It is modeled on the way a small-town banker might deal with a group of customers to whom he has granted lines of credit. The algorithm checks to see if granting the request leads to an unsafe state. If it does, the request is denied. If granting the request leads to a safe state, it is carried out. In Fig. 8.2 (a) we see four customers - *C1*, *C2*, *C3* and C4, each of whom has been granted a certain number of credit units. The banker knows that not all customers will need their maximum credit immediately, so he has reserved only 10 units rather than 22 to service them.

(a) : Safe

	Has	Max
C1	0	6
C2	0	5
C3	0	4
C4	0	7

Free: 10

(b) : Safe

	Has	Max
C1	1	6
C2	1	5
C3	2	4
C4	4	7

Free: 2

(c) : Unsafe

	Has	Max
C1	1	6
C2	2	5
C3	2	4
C4	4	7

Free: 1

Fig. 8.2 : Banker's algorithm example

The customers make loan requests (asking for resources) from time to time for their respective business. At a certain moment, the situation is as shown in Fig. 8.2(b). This state is safe because with two units left, the banker can delay any requests except *C3*'s, thus letting *C3* finish and release all four of his resources. With four units in hand, the banker can let either C4 or C2 have the necessary units, and so on.
Consider a request from C2 for one more unit were granted in Fig. 8.2(b). Then the situation becomes as shown in Fig. 8.2(c), which is unsafe. If all the customers suddenly asked for their maximum loans, the banker could not satisfy any of them, and we would have a deadlock. An unsafe state does not *have* to lead to deadlock, since a customer might not need the entire credit line available, but the banker cannot count on this behavior.

The banker's algorithm considers each request as it occurs, and sees if granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed until later. To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer. If so, those loans are assumed to be repaid, and the customer now closest to the limit is checked, and so on. If all loans can eventually be repaid, the state is safe and the initial request can be granted.



CHECK YOUR PROGRESS

Q1. State whether the following statements are true(T) or false(F):

a) Deadlock can be defined as the temporary blocking of a set of processes that either compete for system resources or communicate with each other Scheduling is done by switching the CPU among processes.

b) Examples of reusable resources include processors, I/O channels, main and secondary memory, devices, and data structures such as files, databases etc.

c) A consumable resource is one that can be created and destroyed.

d) The circular-wait condition can be prevented by defining a linear ordering of resource types.

e) "Abort all deadlocked processes" is not a way to Recovery from Deadlock.

f) Deadlock may involve reusable resources or consumable resources.

Q2. Answer the following by selecting the appropriate option:

a) Two general categories of resources
(i) Reusable and Consumable.
(ii) Preempted and Nonpreempted.
(iii) Both (i) and (ii).
(iv) None of the above.

b) Examples of consumable resources are
(i) Interrupts (ii) Signals
(iii) Messages (iv) All of the above

c) The condition(s) that must be present for a deadlock to occur, are
(i) Mutual Exclusion Condition
(ii) Hold and Wait Condition
(iii) Circular Wait Condition
(iv) All of the above

d) Banker's Algorithm is a
(i) Deadlock Detection Algorithm.
(ii) Deadlock Avoidance Algorithm.
(iii) Both (i) and (ii).
(iv) None of the above.

e) Deadlock prevention guarantees that
(i) deadlock will not occur.
(ii) deadlock will occur.
(iii) Both (i) and (ii).
(iv) None of the above



8.8 LET US SUM UP

- Deadlock is the blocking of a set of processes that either compete for system resources or communicate with each other. The blockage is permanent unless the operating system takes some extraordinary action, such as killing one or more processes or forcing one or more processes to backtrack.
- Deadlock may involve reusable resources or consumable resources. A reusable resource is one that is not depleted or destroyed by use, such as an I/O channel or a region of memory. A consumable resource is one that is destroyed when it is acquired by a process; examples include messages and information in I/O buffers.
- There are three general approaches to dealing with deadlock: prevention, detection, and avoidance.
- Deadlock prevention guarantees that deadlock will not occur, by assuring that one of the necessary conditions for deadlock is not met.
- Deadlock detection is needed if the operating system is always willing to grant resource requests; periodically, the OS must check for deadlock and take action to break the deadlock.
- Deadlock avoidance involves the analysis of each new resource request to determine if it could lead to deadlock, and granting it only if deadlock is not possible.



8.9 FURTHER READING

1. Silberschatz, A., Galvin, P. B., Gagne, G., & Silberschatz, A. (1998). Operating system concepts (Vol. 4). Reading: Addison-wesley.

2. Tanenbaum, A. S., & Woodhull, A. S. (1987). Operating systems: design and implementation (Vol. 2). Englewood Cliffs, NJ: Prentice-Hall.



8.10 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1. (a) False

(b) True

(c) True

(d) True

(e) False

(f) True

Answer to Q2. (a) (i)

(b) (iv)

(c) (iv)

(d) (ii)

(e) (i)



8.11 MODEL QUESTIONS

Q1. Define deadlock.

Q2. What are reusable resource and consumable resource? How are they related with deadlock?

Q3. What are the conditions that must be present for deadlock to occur? Explain in brief.

Q4. Explain deadlock modeling.

Q5. What is the basic difference between deadlock detection and deadlock avoidance?

Q6. Explain the banker's algorithm for deadlock avoidance.

Q7. Write short notes on:

(i) Deadlock Modeling.

(ii) Deadlock Recovery.

(iii) Deadlock Prevention.

(iv) Deadlock Avoidance.

****